# Javaweb

## 定义

在Sun的Java Servlet规范中，对Java Web应用作了这释：

定义：“Java Web应用由一组Servet、HTML页、类、以及其它可以被绑定的资源构成。它可以在各种供应商提供的实现Servlet规范的Serlet容器中运行。”

## 内容

1、Servlet

2、JSP

3、实用类

4、静态的文档如HTML、图片等

5、描述web应用的信息（web.xml）

## 软件的架构：

1、C/S（客户端/服务器）(client/server)

1、一般我们使用的软件都是C/S结构

2、比如系统的中软件qq、360、office

3、C表示客户端，用户通过客户端来使用软件

4、S表示服务器，服务器负责软件处理软件的业务逻辑

5、特点：1）软件使用前必须的安装，软件更新时，服务器和客户端同时更新。

2）C/S架构的软件不能跨平台使用。

3）C/S架构的软件客户端和服务器通信采用的是自由协议，相对来说比较安全。

2、B/S 浏览器/服务器

1、B/S本质上也是C/S，只不过B/S架构的软件，使用浏览器作为软件的客户端。

2、B/S架构软件通过使用浏览器访问网页的形式，来使用软件。

3、比如：京东，淘宝、12306.

4、特点：软件不需要安装，直接使用浏览器访问指定的网页。

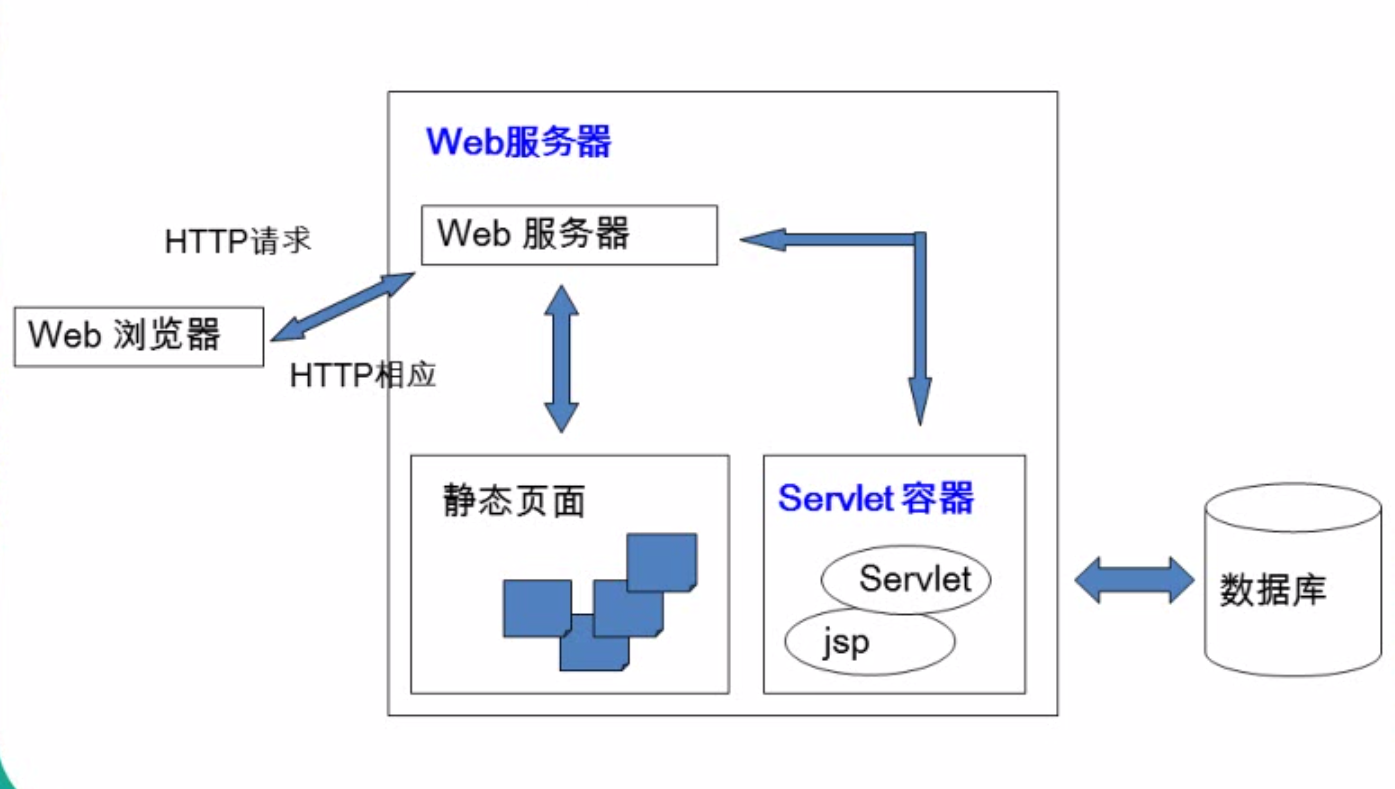
2）软件更新时，客户端不跟新

3）软件可以跨平台。只要系统有浏览器，就可以使用。

4）B/S架构的软件，客户端和服务器之间通信采用通用的Http协议。

5)相对来说不安全。可http +s 加了密

## 3、web资源介绍运行原。



Web浏览器发送一个请求到web服务器，web服务器会解析这个请求是请求动态资源还是静态资源。如果是静态页面，会直接找到并并返回。如果是动态资源，则由servlet容器中的servlet进行解析，解析的过程可能为与数据库发生关系，然后将servlet返回的结果由jsp处理，返回一个html形式返回。Servlet类访问数据库采用JDBC。

重点：Servlet容器和在Servlet容器中写servlet和jsp，可以被servlet调用。

### web浏览器

浏览器是指 Web 服务的客户端浏览程序。 它可以向服务器发送各种请求， 并对从服务器中返回的各种信息（包括文本、超文本、音频、视频等各种数据）进行解释、显示和播放。

### 2、web服务器

浏览器与服务器的关系可谓是 “唇齿相依飞浏览器发送请求， 服务器处理请求井将结果返回给浏览器显示。Web 服务器的种类繁多，目前比较流行的有 WebSphere、WebLogic、Tomcat 等。它们的配置、启动方式各不相同，也各自有其优缺点。

Web 服务器和 JSP/Servlet 容器，通常这两者会集于一身。

#### html:静态资源

#### 2、Servlet容器

Servlet容器：也称为JavaWeb应用容器，或者Servlet/JSP容器。

作用：为JavaWeb应用提供运行时环境，它负责管理Servlet和JSP的生命周期，以及管理它们的共享数据。JSP/Servlet 容器的基本功能是把动态资源 转换成静态资源

1. 作用：Web 服务器的作用是接收客户端的请求，给客户端作出响应
2. Java Servlet（Java服务器小程序）是一个基于Java技术的Web组件，运行在服务器端，它由Servlet容器所管理，用于生成动态的内容。 Servlet是平台独立的Java类，编写一个Servlet，实际上就是按照Servlet规范编写一个Java类。Servlet被编译为平台独立 的字节码，可以被动态地加载到支持Java技术的Web服务器中运行。   
   Servlet容器也叫做Servlet引擎，是Web服务器或应用程序服务器的一部分，用于在发送的请求和响应之上提供网络服务，解码基于 MIME的请求，格式化基于MIME的响应。Servlet没有main方法，不能独立运行，它必须被部署到Servlet容器中，由容器来实例化和调用 Servlet的方法（如doGet()和doPost()），Servlet容器在Servlet的生命周期内包容和管理Servlet。在JSP技术 推出后，管理和运行Servlet/JSP的容器也称为Web容器。

### 3、数据库

### 4、静态资源和动态资源的区别:

静态网页：直接由HTML标记语言编写的网页通常称为静态网页。所谓静态网页与动态网页，是基于访问网页时页面的内容有无变化而言的，与页面的视觉效果没有关系。因为动态的视觉效果大多是通过JavaScript或其他jQuery技术实现的，与动态网页技术没有直接必然的联系。

静态网站特点：

　 1. 静态网站是最初的建站方式，浏览者所看到的每个页面是建站者上传到服务器上的一个 html （ htm ）文件，这种网站每增加、删除、修改一个页面，都必须重新对服务器的文件进行一次下载上传。网页内容一经发布到网站服务器上，无论是否有用户访问，每个静态网页的内容都是保存在网站服务器上的，也就是说，静态网页是实实在在保存在服务器上的文件，每个网页都是一个独立的文件；

　 2. 静态网页的内容相对稳定，因此容易被搜索引擎检索；

　 3. 静态网页没有数据库的支持，在网站制作和维护方面工作量较大，因此当网站信息量很大时完全依靠静态网页制作方式比较困难；

　 4. 静态网页的交互性较差，在功能方面有较大的限制。

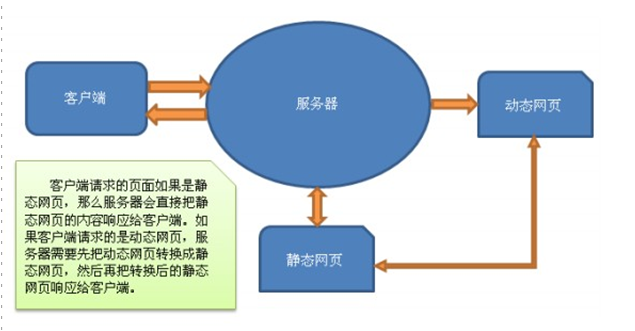
动态资源：一般客户端请求的动态资源，先将请求交于web容器，web容器连接数据库，数据库处理数据之后，将内容交给web服务器，web服务器返回给客户端解析渲染理。

动态网页中的变化内容大部分是来自于数据库中数据的变化。它是通过增加、删除、修改、 查找数据库中存储的数据来显示内容的变化。例如在微博中发布一条微博后，查看微博时，会将所发的微博即时显示出来，这在静态网页中是无法完成的。

目前比较流行的动态网页技术主要包含：ASP、PHP以及JSP。

1. 交互性：网页会根据用户的要求和选择而动态地改变和响应，浏览器作为客户端，成为一个动态交流的桥梁，动态网页的交互性也是今后 Web 发展的潮流。

2. 自动更新：即无须手动更新 HTML 文档，便会自动生成新页面，可以大大节省工作量。

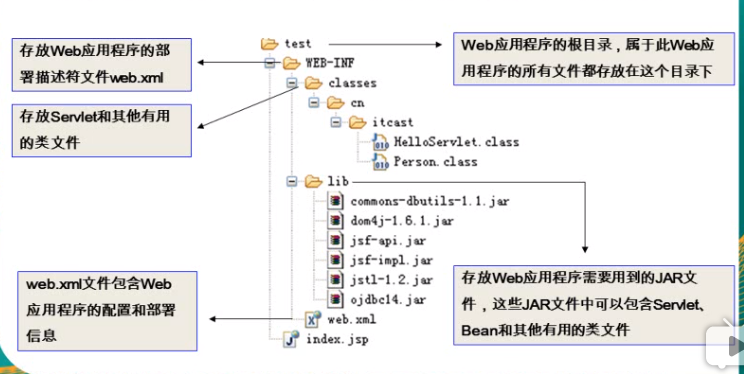
3. 因时因人而变：即当不同时间、不同用户访问同一网址时会出现不同页面。

## 访问web资源

打开浏览器，输入URL：协议名 :// 域名 :端口 / 路径

例如： http://www.itcast.cn:80/index.html

## Web程序结构



一个web应用程序是由一组Servlet，HTML页面，类，以及其它的资源组成的运行在web服务器上的完整的应用程序，以一种结构化的有层次的目录形式存在组成web应用程序的这些文件要部署在相应的目录层次中，根目录代表整个web应用程序的”根”通常将web应用程序的目录放在webapps目录下，在webapps目录下的每一个子目录都是一个独立的web应用程序，子目录的名字就是web应用程序的名字，也就是web应用程序的“根”。用户通过web应用程序的”根”来访问web应用程序中的资源

## web容器（Servlet 引擎）

1、可以创建servlet，并调用servlet的相关声明周期方法。

2、JSP，filter，listener，tag…..都是在servlet容器中运行，并被调永相应的生命周期方法。

主流的servlet容器：

Tomcat（ Apache）：当前应用最广的JavaWeb 服务器；JSP/Servlet容器

注意：Tomcat 不支持JavaEE规范；

JBoss（ Redhat 红帽）：支持 JavaEE，应用比较广；EJB容器；

GlassFish（ Orcale）： Oracle 开发 JavaWeb 服务器，应用不是很广；

Resin（ Caucho）：支持 JavaEE，应用越来越广；

Weblogic （ Orcale）：要钱的！支持JavaEE，适合大型项目；

Websphere（ IBM ）：要钱的！支持JavaEE，适合大型项目；

## 第一个web程序

一个web应用程序是由一组Servlet，HTML页面，类，以及其它的资源组成的运行在web服务器上的完整的应用程序，以一种结构化的有层次的目录形式存在组成web应用程序的这些文件要部署在相应的目录层次中，根目录代表整个web应用程序的”根”

通常捋 web应用程序的目录放在webapps目录下，在webapps目录下的每一个子目录都是一个独立的web应用程序，子目录的名字就是web应用程序的名字，也就是web应用程序的“根”。用户通过web应用程序的”根”来访问web应用程序中的资源。

### 在Eclipse新建一个Java Project

### 在Java项目下创建WEB开发的目录结构

-src(源文件)

-WebContent

-WEB-INF

-classes (编译后的 class 文件必须放在该目录下.)

-lib

-web.xml (从apache-tomcat-6.0.16\webapps\docs\WEB-INF 中复制过来,可以不做修改)当前web的配置信息，servlet，fileter等。

html 页面

JSP 页面(html页面中可以写java代码)

图片...

Hello：应用目录hello 就是应用的名称；

index.html ：应用资源。应用下可以有多个资源，例如 css、js、html 、jsp ，也可以把资源放到文件夹中。

例如：hello\html\index.html

这时访问URL为：http://localhost:8080/hello/html/index.html ；

WEB-INF：这个目录名称必须是大写，这个目录下的东西是无法通过浏览器直接访问的，也就是说放到这里的东西是安全的。

lib：存放jar包的目录。

web.xml: 应用程序的部署描述符文件，可以在该文件中对应用进行配置.

### 在src下新建一个person类。

package com.atguigu.test;

public class Person {

public String getPersonInfo(){

return "person info...";

}

### 手动的将person类对应的class文件（含包）复制到classes目录中。

注：可以通过修改的默认输出目录达到把编译好的class放到classes目录下：

右键-build path-configure build path-browse

### 在WebContent目录下新建一个JSP文件。

<%@page import="com.atguigu.test.Person"%>

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

<%

Person person = new Person();

System.out.print(person.getPersonInfo());

%>

### 把WebContent目录复制到Tomcat的webapps目录下，并修改web项目的名称。

### 配置外部应用：项目不在tomcat的webapps文件夹下。

例如我们把上面写的 hello 应用从 webapps目录中剪切到C盘下，即C:/hello现在hello 这个 Web 应用已经不在 Tomcat 中了，这时我们需要 在 tomcat 中配置外部应用的位置，配置的方式一共有两种：

在tomcat文件目录 conf 目录下依次创建 catalina\localhost 目录，然后在 localhost 目录下为 test 这个Web应用程序建立 test.xml 文件，编辑这个文件输入以下内容：

<?xml version="1.0" encoding="UTF-8"?>

<Context docBase="E:\\Java\\Source\\atguigu\\java-1\\firstWebApp\\WebContent"

reloadable="true"/>

reloadable：当修改了 docBase 下的classes目录下的class文件时，tomcat 服务器是否会重新加载WEB应用。当将reloadable设为false时，修改了class 文件，不会重新加载。

访问：http://localhost:8080/ test/test.jsp

访问过程，访问该网址，找到test文件夹，然后在该文件夹中找到docBase的物理地址，然后物理地址找到对应WebContent内容。

# HTTP

## HTTP简介：

HTTP(hypertext transport protocol)即超文本传输协议。这个协议详细规定了浏览器和万维网服务器之间互相通信的规则。它是TCP/IP协议集中的一个应用层协议，用于定义WEB浏览器与WEB服务器之间交换数据的过程以及数据本身的格式。

## HTTP的格式

浏览器发出的请求消息和web服务器回送的响应消息都叫HTTP消息,HTTP消息有一定严格规定的格式. 浏览器提交给web服务器的form表单内容和从web服务器上获取的网页内容仅仅是HTTP消息的一部分数据,浏览器与web服务器传递的信息还包含一般用户所看不到的一些其他"隐藏"的信息。

浏览器发送给服务器的内容就这个格式的，如果不是这个格式，服务器将无法解读！在HttpI协议中，请求有很多种请求方法，其中最为常用的就是get和post。

格式：

请求首行；

请求头信息；

空行；

请求体。

### get请求



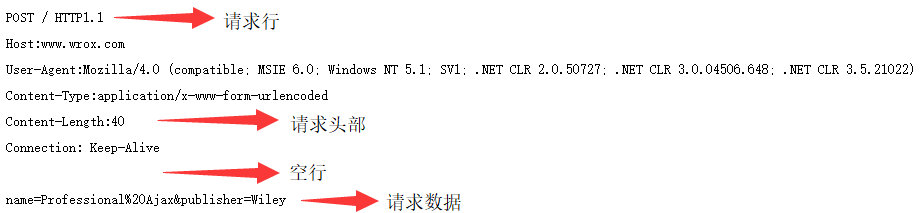
第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的HTTP版本。

第二部分：请求头部，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息。

第三部分：空行，请求头部后面的空行是必须的。

第四部分：请求数据也叫主体，可以添加任意的其他数据。

### post请求



第一部分：请求行，第一行明了是post请求，以及http1.1版本。

第二部分：请求头部，第二行至第六行。

第三部分：空行，第七行的空行。

第四部分：请求数据，第八行。

## 请求协议HTTP消息的格式和响应协议

## GET请求

1、在浏览器地址栏中输入某个URL地址或单击网页上的一个超链接时，浏览器发出的HTTP请求消息的请求方式为GET。

2、如果网页中的<form>表单元素的method 属性被设置为了“GET”，浏览器提交这个FORM表单时生成的HTTP请求消息的请求方式也为GET。如果没有设置method属性时，默认为get请求。

3、使用GET请求方式给WEB服务器传递参数的格式：

<http://www.atguigu.com/counter.jsp?name=lc&password=123>

4、使用GET传递数据一遍在1KB以下。

|  |
| --- |
| GET /counter.jsp?name=lc&password=123 HTTP/1.1 （请求首行， get 没有请求体）  Host: localhost  User-Agent: Mozilla/ 5.0 (Windows NT 5.1; rv:5.0) Gecko/ 20100101 Firefox/ 5.0  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  Accept-Language: zh-cn,zh;q=0.5  Accept-Encoding: gzip, deflate  Accept-Charset: GB2312,utf-8;q=0.7,\*;q=0.7  Connection: keep-alive  Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D9（请求头类似 map） |

解释：

GET hello/index.jsp HTTP/1.1 ：请求，请求服务器路径为 /counter.jsp?name=lc&password=123 ，协议为1.1。

Host:localhost ：请求的主机名为localhost ；

User-Agent: Mozilla/ 5.0 (Windows NT 5.1 ; rv:5.0) Gecko/ 20100101 Firefox/5.0： 与浏览器和OS相关的信息。有些网站会显示用户的系统版本和浏览器版本信息，这都是通过获取。

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8 ： 告诉服务器，当前客户端可以接收的文档类型，其实这里包含了\*/\* ，就表示什么都可以接收；

Accept-Encoding: gzip, deflate ：支持的压缩格式。数据在网络上传递时，可能服务器会把数据压缩后再发送；

Accept-Charset: GB2312,utf-8;q=0.7,\*;q=0.7 ：客户端支持的编码；

Connection: keep-alive ：客户端支持的链接方式，保持一段时间链接，默认为3000ms ；

Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98因为不是第一次访问这个地址， 所以会在请求中把上一次服务器响应中发送过来的Cookie 在请求中一并发送去过；这个Cookie 的名字为JSESSIONID然后在讲会话是讲究它！

## POST请求

1、POST 请求方式主要用于向WEB服务器端程序提交FORM表单中的数据：form 表单的 method 置POST。

2、POST方式将各个表单字段元素及其数据作为HTTP消息的实体内容发送给WEB服务器，传送的数据量要比使用GET方式传送的数据量大得多。

例如：

表单：

|  |
| --- |
| 为了演示POST请求，我们需要修改index.jsp 页面，即添加一个表单：  < form action= ""method="post">  关键字：<input type = "text" name ="id" />  <input type = "text" name ="name" />  <input type ="submit" value="提交"/>  </form> |

POST

|  |
| --- |
| POST:  POST /hello/index.jsp HTTP/1.1  Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/msword, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/x-ms-application, application/x-ms-xbap,  application/vnd.ms-xpsdocument, application/xaml+xml, \*/\*  Referer: <http://localhost:8080/hello/index.jsp>  Accept-Language: zh-cn,en-US;q=0.5  User-Agent: Mozilla/ 4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/ 4.0; InfoPath.2; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)  Content-Type: application/x-www-form-urlencoded  Accept-Encoding: gzip, deflate  Host: localhost:8080  Content-Length: 13  Connection: Keep-Alive  Cache-Control: no-cache  Cookie: JSESSIONID=E365D980343B9307023A1D271CC48E7D  id=1&name=sx //请求头与请求体之间空一行， |

解释：Referer: http://localhost:8080/hello/index.jsp ：请求来自哪个页面，例如你在百度上点击链 接到了这里，那么 Referer:http://www.baidu.com ；如果你是在浏览器的地址栏中直接输入 的地址，那么就没有Referer 这个请求头了；

Content-Type: application/x-www-form-urlencoded ：表单的数据类型，说明会使用url 格式编码数据；url编码的数据都是以“%”为前缀，后面跟随两位的16进制，例如“传智”这两个字使用UTF-8 的 url 编码用为“ %E4%BC%A0%E6%99%B”A；

Content-Length:13 ：请求体的长度，这里表示 13个字节。

id=1&name=sx ：请求体内容！hello 是在表单中输入的数据，keyword 是表单字段的名字。

## refer请求头

Referer请求头是比较有用的一个请求头，它可以用来做统计工作，也可以用来做防盗链 。

统计工作：我公司网站在百度上做了广告，但不知道在百度上做广告对我们网站的访问量是否有影响，那么可以对每个请求中的Referer 进行分析，如果Referer 为百度的很多，那么说明用户都是通过百度找到我们公司网站的。防盗链 ：我公司网站上有一个下载链接，而其他网站盗链了这个地址，例如在我网站上的

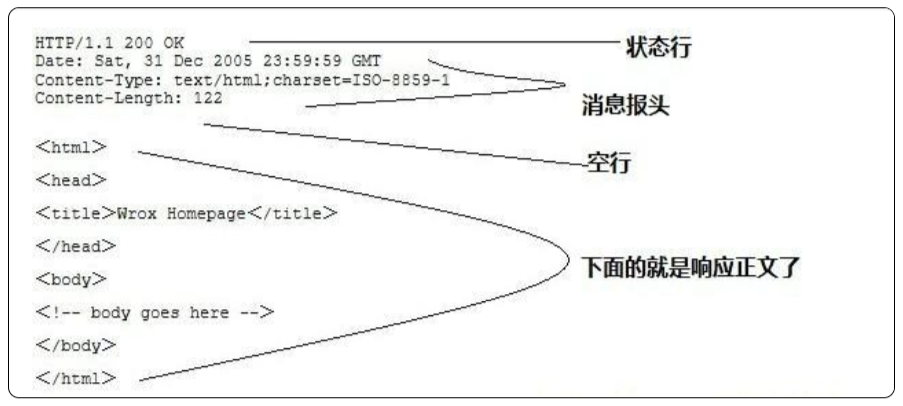
index.html 页面中有一个链接， 点击即可下载JDK7.0，但有某个人的微博中盗链了这个资源，它也有一个链接指向我们网站的JDK7.0，也就是说登录它的微博，点击链接就可以从我网站上下载JDK7.0， 这导致我们网站的广告没有看，但下载的却是我网站的资源。这时可以使用Referer 进行防盗链，在资源被下载之前，我们对Referer 进行判断，如果请求来自本网站，那么允许下载，如果非本网站，先跳转到本网站看广告，然后再允许下载。

## 响应协议

一般情况下，服务器接收并处理客户端发过来的请求后会返回一个HTTP的响应消息。

### 响应格式

HTTP响应也由四个部分组成，分别是：状态行、消息报头、空行和响应正文。



第一部分：状态行，由HTTP协议版本号， 状态码， 状态消息 三部分组成。

第一行为状态行，（HTTP/1.1）表明HTTP版本为1.1版本，状态码为200，状态消息为（ok）

第二部分：消息报头，用来说明客户端要使用的一些附加信息。

第二行和第三行和第四行为消息报头，Date:生成响应的日期和时间；Content-Type:指定了MIME类型的HTML(text/html),编码类型是ISO-8859-1

第三部分：空行，消息报头后面的空行是必须的。

第四部分：响应正文，服务器返回给客户端的文本信息。

空行后面的html部分为响应正文。

响应内容是由服务器发送给浏览器的内容，浏览器会根据响应内容来显示。

|  |
| --- |
| HTTP/1.1 200 OK  Server: Apache-Coyote/1.1  Content-Type: text/html;charset=UTF-8 //有些头可以有多个值。  Content-Length: 724  Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello  Date: Wed, 25 Sep 2012 04:15:03 GMT  上面的响应内容不会显示，只会显示下面的。  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <base href="http://localhost:8080/hello/">    <title>My JSP 'index.jsp' starting page</title>  <meta http-equiv="pragma" content="no-cache">  <meta http-equiv="cache-control" content="no-cache">  <meta http-equiv="expires" content="0">  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">  <meta http-equiv="description" content="This is my page">  <!--  <link rel="stylesheet" type="text/css" href="styles.css">  -->  </head>    <body>  <form action="" method="post">  关键字：<input type="text" name="keyword"/>  <input type="submit" value="提交"/>  </form>  </body>  </html> |

HTTP/1.1 200 OK：响应协议为HTTP1.1，状态码为200，表示请求成功，OK是对状态码的解释；

Server: Apache-Coyote/1.1：服务器的版本信息；

Content-Type: text/html;charset=UTF-8：响应体使用的编码为UTF-8；

Content-Length: 724：响应体为724字节；

Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello：响应给客户端的Cookie；

Date: Wed, 25 Sep 2012 04:15:03 GMT：响应的时间，这可能会有8小时的时区差；

### 响应码

响应头对浏览器来说很重要，它说明了响应的真正含义。例如200表示响应成功了，302表示重定向，这说明浏览器需要再发一个新的请求。

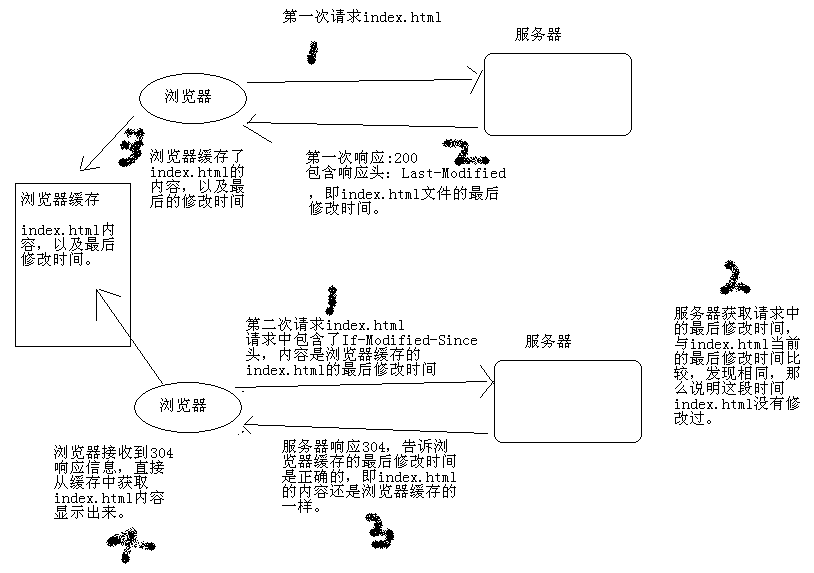
 200：请求成功，浏览器会把响应体内容（通常是html）显示在浏览器中；

 404：请求的资源没有找到，说明客户端错误的请求了不存在的资源；客户端错误。

 500：请求资源找到了，但服务器内部出现了错误；服务器端错误。

 302：重定向，当响应码为302时，表示服务器要求浏览器重新再发一个请求，服务器会发送一个响应头Location，它指定了新请求的URL地址；

304：当用户第一次请求index.html时，服务器会添加一个名为Last-Modified响应头，这个头说明了index.html的最后修改时间，浏览器会把index.html内容，以及最后响应时间缓存下来。当用户第二次请求index.html时，在请求中包含一个名为If-Modified-Since请求头，它的值就是第一次请求时服务器通过Last-Modified响应头发送给浏览器的值，即index.html最后的修改时间，If-Modified-Since请求头就是在告诉服务器，我这里浏览器缓存的index.html最后修改时间是这个，您看看现在的index.html最后修改时间是不是这个，如果还是，那么您就不用再响应这个index.html内容了，我会把缓存的内容直接显示出来。而服务器端会获取If-Modified-Since值，与index.html的当前最后修改时间比对，如果相同，服务器会发响应码304，表示index.html与浏览器上次缓存的相同，无需再次发送，浏览器可以显示自己的缓存页面，如果比对不同，那么说明index.html已经做了修改，服务器会响应200。



响应头：

 Last-Modified：最后的修改时间；

请求头：

 If-Modified-Since：把上次请求的index.html的最后修改时间还给服务器；

状态码：304，比较If-Modified-Since的时间与文件真实的时间一样时，服务器会响应304，而且不会有响正文，表示浏览器缓存的就是最新版本！、

5开头全部是服务器段错误，4开头全部为客户端错误，2开头全部为成功。

### 其他响应头

告诉浏览器不要缓存的响应头：

 Expires: -1；过期时间

 Cache-Control: no-cache；不缓存

 Pragma: no-cache；不缓存

自动刷新响应头，浏览器会在3秒之后请求http://www.itcast.cn：

 Refresh: 3;url=http://www.itcast.cn

注意和Referer的区别，Referer是防盗链，Refresh:是自动刷新

Html页面中模拟相应头 ： meta标签

<metahttp-equiv="Refresh"content="5;url=http://localhost:8080/day08\_1/index.jsp">

### HTML中指定响应头（注意看HTML中的mate标签）

在HTMl页面中可以使用<meta http-equiv="" content="">来指定响应头，例如在index.html页面中给出<meta http-equiv="Refresh" content="3;url=http://www.itcast.cn">，表示浏览器只会显示index.html页面3秒，然后自动跳转到http://www.itcast.cn。

简易总结：

动态资源：需要转换成静态资源后再响应给客户端，例如：jsp、servlet，其他语言的动态资源有：asp、php

静态资源：无需转发即可直接响应给客户端，例如：html、css、javascript

## HTTP工作原理

HTTP协议定义Web客户端如何从Web服务器请求Web页面，以及服务器如何把Web页面传送给客户端。HTTP协议采用了请求/响应模型。客户端向服务器发送一个请求报文，请求报文包含请求的方法、URL、协议版本、请求头部和请求数据。服务器以一个状态行作为响应，响应的内容包括协议的版本、成功或者错误代码、服务器信息、响应头部和响应数据。

以下是 HTTP 请求/响应的步骤：

1、客户端连接到Web服务器

一个HTTP客户端，通常是浏览器，与Web服务器的HTTP端口（默认为80）建立一个TCP套接字连接。例如，http://www.oakcms.cn。

2、发送HTTP请求

通过TCP套接字，客户端向Web服务器发送一个文本的请求报文，一个请求报文由请求行、请求头部、空行和请求数据4部分组成。

3、服务器接受请求并返回HTTP响应

Web服务器解析请求，定位请求资源。服务器将资源复本写到TCP套接字，由客户端读取。一个响应由状态行、响应头部、空行和响应数据4部分组成。

4、释放连接TCP连接

若connection 模式为close，则服务器主动关闭TCP连接，客户端被动关闭连接，释放TCP连接;若connection 模式为keepalive，则该连接会保持一段时间，在该时间内可以继续接收请求;

5、客户端浏览器解析HTML内容

客户端浏览器首先解析状态行，查看表明请求是否成功的状态代码。然后解析每一个响应头，响应头告知以下为若干字节的HTML文档和文档的字符集。客户端浏览器读取响应数据HTML，根据HTML的语法对其进行格式化，并在浏览器窗口中显示。

# Tomcat

## Tomcat简介

Tomcat 服务器是一个免费的开放源代码的Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好Apache 服务器，可利用它响应HTML（标准通用标记语言下的一个应用）页面的访问请求。实际上Tomcat 部分是Apache 服务器的扩展，但它是独立运行的，所以当你运行tomcat 时，它实际上作为一个与Apache 独立的进程单独运行的。

诀窍是，当配置正确时，Apache 为HTML页面服务，而Tomcat 实际上运行JSP 页面和Servlet。另外，Tomcat和IIS等Web服务器一样，具有处理HTML页面的功能，另外它还是一个Servlet和JSP容器，独立的Servlet容器是Tomcat的默认模式。不过，Tomcat处理静态HTML的能力不如Apache服务器。目前Tomcat最新版本为9.0

作用：所调服务器其实就是代码编写的一个可以根据用户请求实时的调用执行对应的逻辑代码的一个容器在普通用户看来就是一个安装程序。我们只需要将服务器在操作系统上进行安装,并将我们事先编写好的逻辑处理代码根据规则放到服务器的指定位置.启动服务器,那么服务器就自动的会根据接收到请求调用并执行对象的逻辑代码进行处理。

## Tomcat安装和文件介绍

### 官方下载地址：

http://tomcat.apache.org

### Tomcat分为安装版和解压版：（目录的名称不能是中文）

A.安装版：一台电脑上只能安装一个Tomcat；

B.解压版：无需安装，解压即可用，解压份数无限制，使用比较方便，我们选择解压版。

### 启动Tomcat服务器：

A.启动前必须配置JAVA\_HOME环境变量。因为Tomcat程序本身是用java写的，需要用到JDK。

B. 2、配置Tomcat环境变量:

右击【计算机】---【属性】-----【高级系统设置】---【环境变量】

1，新建用户变量名：CATALINA\_BASE，变量值: D:\apache-tomcat-8.0.39 (可选）

1，新建用户变量名：TOMCAT\_HOME，变量值:D:\apache-tomcat-8.0.39 （可选

2，新建用户变量名：CATALINA\_HOME，变量值：D:\apache-tomcat-8.0.39 （必须）

3.置环境系统变量Path，添加变量值：%CATALINA\_HOME%\lib;%CATALINA\_HOME%\bin

C.进入到Tomcat解压目录下的bin\startup.bat 双击可启动Tomcat服务器

D.进入到Tomcat解压目录下的bin\shutdown.bat 双击即可关闭Tomcat服务器

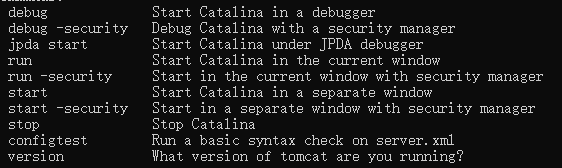
E.启动完成后可以在浏览器地址栏输入：http://localhost:8080 来检查Tomcat是否启动成功，如果看到如下页面。则恭喜你成功了。

F. 若已经启动了一个Tomcat 应用，若再启动同一个Tomcat应用，会抛出异常：

java.net.BindException:Address already in use:JVM\_Bind:8080因为端口已经被占用了。

G：.阅读startup.bat 文件，发现启动tomcat 服务器的是catalina.bat 文件

在命令行窗口直接输入catalina，提示如下：



Run：在catalina同一个命令行窗口下启动服务器。

Start：开启一个新窗口启动服务器，有下表闪，等待浏览器的访问。

Stop：关闭服务器

### 配置端口号

Tomcat的默认端口号为8080，可以在server.xml 中可以更改服务端口和改变 Web 默认的访问目录。

（1）修改端号

<Connector port=“8080”(端口号) protocol＝”HTTP/1.1” connectionTimeout=”20000” reclirectPort=”8443” URIEncoding=”UTF-8”／〉

### 在任意文件夹启动Tomcat

1. 把 D:\apache-tomcat-6.0.16\bin 添加到 path 下

添加成功后, 若在任意目录下通过 startup 启动 Tomcat 服务器, 会有提示:

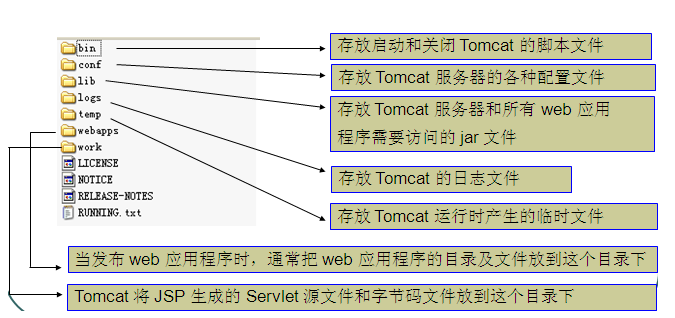
The CATALINA\_HOME environment variable is not defined correctly This environment variable is needed to run this program

即: 要求设置 CATALINA\_HOME 这个环境变量

2. 设置 CATALINA\_HOME 环境变量:、

通过阅读 startup.bat 批处理文件, 可知 CATALINA\_HOME 环境变量下有一个 bin 目录,而 bin 目录又有一个 catalina.bat 文件, 由此可知 CATALINA\_HOME 指向的应该是 Tomcat 安装的根目录。

### 文件介绍：



1)bin：该目录下存放的是二进制可执行文件会有startup.bat和shutdown.bat文件（其余文件无需理会），startup.bat用来启动Tomcat，但需要先配置JAVA\_HOME环境变量才能启动，shutdawn.bat用来停止Tomcat；

2)conf：这是一个非常非常重要的目录，这个目录下有四个最为重要的文件：

server.xml：配置整个服务器信息。例如修改端口号，添加虚拟主机等；以后会详细介绍这个文件；

tomcat-users.xml：存储tomcat用户的文件，这里保存的是tomcat的用户名及密码，以及用户的角色信息。可以按着该文件中的注释信息添加tomcat用户，然后就可以在Tomcat主页中进入Tomcat Manager页面了；

web.xml：部署描述符文件，这个文件中注册了很多MIME类型，即文档类型。这些MIME类型是客户端与服务器之间说明文档类型的，如用户请求一个html网页，那么服务器还会告诉客户端浏览器响应的文档是text/html类型的，这就是一个MIME类型。客户端浏览器通过这个MIME类型就知道如何处理它了。当然是在浏览器中显示这个html文件了。但如果服务器响应的是一个exe文件，那么浏览器就不可能显示它，而是应该弹出下载窗口才对。MIME就是用来说明文档的内容是什么类型的！

context.xml：对所有应用的统一配置，通常我们不会去配置它。

3)lib：Tomcat的类库，里面是一大堆jar文件。如果需要添加Tomcat依赖的jar文件，可以把它放到这个目录中，当然也可以把应用依赖的jar文件放到这个目录中，这个目录中的jar所有项目都可以共享之，但这样你的应用放到其他Tomcat下时就不能再共享这个目录下的Jar包了，所以建议只把Tomcat需要的Jar包放到这个目录下，各个应用需要的jar各个应用自己管理。

4)logs：这个目录中都是日志文件，记录了Tomcat启动和关闭的信息，如果启动Tomcat时有错误，那么异常也会记录在日志文件中。

5)temp：存放Tomcat的临时文件，这个目录下的东西可以在停止Tomcat后删除！

6)webapps：存放web项目的目录，其中每个文件夹都是一个项目；如果这个目录下已经存在了目录，那么都是tomcat自带的。项目。其中ROOT是一个特殊的项目，在地址栏中没有给出项目目录时，对应的就是ROOT项目。http://localhost:8080/examples，进入示例项目。其中examples就是项目名，即文件夹的名字。

7)work：运行时生成的文件，最终运行的文件都在这里。通过webapps中的项目生成的！可以把这个目录下的内容删除，再次运行时会生再次生成work目录。当客户端用户访问一个JSP文件时，Tomcat会通过JSP生成Java文件，然后再编译Java文件生成class文件，生成的java和class文件都会存放到这个目录下。

8)LICENSE：许可证。

9)NOTICE：说明文件。

## tomcat功能实现

### 案例一

功能：实现向网站发送请求，然后网站返回响应数据。

|  |
| --- |
| **public class** Client {  **public static void** main(String[] args) {  *//1、建立一个socket连接，连接sx的80端口* Socket socket = **null**;  InputStream is = **null**;  OutputStream ops = **null**;   **try** {  *//1、建立一个socket连接，连接sx的80端口* socket = **new** Socket(**"www.sx.cn"**, 80);  *//2、获取对象的输出流对象* is = socket.getInputStream();  *//3、获取到输入流对象* ops = socket.getOutputStream();  *//4、将http协议的请求部分发送到服务端/subject/about/index.html  //发送请求行* ops.write(**"GET / HTTP/1.1\n"**.getBytes());  *//发送请求头* ops.write(**"HOST:www.baidu.com\n"**.getBytes());  *//请求头与请求体之间有一个换行，必须发过去* ops.write(**"\n"**.getBytes());  *//5、读取来自服务器端的数据打印到控制台* **int** i = is.read();  **while** (i != -1) {  System.***out***.print((**char**) i);  i = is.read();  }  } **catch** (IOException e) {  e.printStackTrace();  }**finally** {  **try** {  ops.close();  } **catch** (IOException e) {  e.printStackTrace();  }  **try** {  is.close();  } **catch** (IOException e) {  e.printStackTrace();  }  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } } } |



### 案例二

从浏览器的地址栏输入localhost:8080，向本机的8080端口发起请求，服务端向客户端响应回去HTTP协议的响应部分

|  |
| --- |
| **public class** server {  **public static void** main(String[] args) {  ServerSocket serverSocket = **null**;  Socket socket = **null**;  OutputStream outputStream = **null**;  **try** {  *//1、创建ServerSockert 对象，监听本机的8080端口号* serverSocket = **new** ServerSocket(8080);  **while** (**true**){  *//2、等待客户端的请求获取和客户端对应的socket对象* socket = serverSocket.accept();  *//3、通过获取的socket对象获取到输出流对象* outputStream = socket.getOutputStream();  *//4、通过获取到的输出流对象将http协议的响应部分发送到客户端  //4.1发送状态行* outputStream.write(**"HTTP/1.1 200 OK\n"**.getBytes());  *//4.2发送响应头* outputStream.write(**"Content-Type:text/html;charset=utf-8\n"**.getBytes());  outputStream.write(**"Server:Apach-Coyote/1.1\n"**.getBytes());*//响应的服务器  //5、换行* outputStream.write(**"\n\n"**.getBytes());  *//6、返送响应体* StringBuffer buffer = **new** StringBuffer();  buffer.append(**"<html>"**);  buffer.append(**"<head><title>我是标题</title></head>"**);  buffer.append(**"<body>"**);  buffer.append(**"<h1>传递的内容</h1>"**);  buffer.append(**"</body>"**);  buffer.append(**"</html>"**);  outputStream.write(buffer.toString().getBytes());  outputStream.flush();  }  } **catch** (IOException e) {  e.printStackTrace();  }**finally** {  *//释放资源* **try** {  outputStream.close();  } **catch** (IOException e) {  e.printStackTrace();  }  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } } |

### 实现tomcat版本

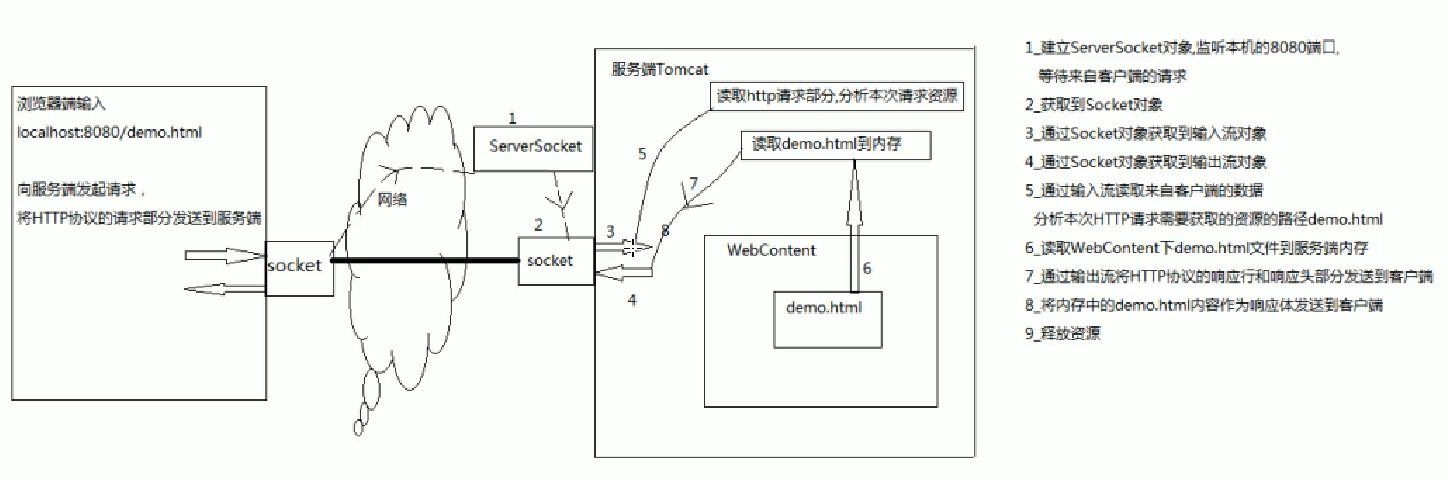
案例需求

1、在WebContent下发布静态资源demo.html，demo02.html

2、启动Tomcat服务器

3、当客户端对服务端发起不同的请求，localhost:8080/demo.html

4、服务端可以将对应的html页面响应到客户端



1、准备功做

1.定义静态变量WEB.ROOT，用于存放WebContent目录的绝对路径。

2.定义静态变量url，存放本次请求服务端的静态资源的名称。

|  |
| --- |
| **public class** server {  *//定义一个变量，存放服务端webcontent目录的绝对路径  //getProperty:System下的方法，用来获取系统的属性* **public static** String *WEB\_Root* = System.*getProperty*(**"user.dir"**)+**"\\"**+**"WebContent"**;  *//定义静态变量，用户存放本次请求的静态页面的名称* **private** String **url** = **""**; |

2、

|  |
| --- |
| **public static void** main(String[] args) {  ServerSocket serverSocket = **null**;  Socket socket = **null**;  OutputStream outputStream = **null**;  InputStream inputStream = **null**;  **try** {  *//1、创建serversocket，监听本机器的80端口，等待客户端的请求* serverSocket = **new** ServerSocket(8080);  **while** (**true**){  *//获取到客户端对应的socket* socket = serverSocket.accept();  *//获取到输入流对象* inputStream = socket.getInputStream();  *//获取到输出流对象* outputStream = socket.getOutputStream();  *//获取http协议的请求部分，截取客户端要访问的资源名称  parse*(inputStream);  *//读取访问的静态资源，并将资源返回给客户端  sendStaticResource*(outputStream);  }  } **catch** (IOException e) {  e.printStackTrace();  }**finally** {  **try** {  inputStream.close();  } **catch** (IOException e) {  e.printStackTrace();  }  **try** {  outputStream.close();  } **catch** (IOException e) {  e.printStackTrace();  }  **try** {  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

//获取http协议的请求部分，截取客户端要访问的资源名称

|  |
| --- |
| *//获取http协议的请求部分，截取客户端要访问的资源名称* **private static void** parse(InputStream is) **throws** IOException {  *//定义各个变量，存放htpp存谢的请求部分数据* StringBuffer content = **new** StringBuffer(2048);  *//定义一个数组，存放http协议的请求部分数据* **byte**[] buffer = **new byte**[2048];  *//定义一个变量i，代表读取数据到数组中之后，数据量的大小* **int** i = -1;  *//读取客户端发送过来的数据，将数据读取到字节数组buffer中，i读取数据量的大小* i = is.read(buffer);  *//遍历字节数组，将数组中的数据追加到content变量中* **for** (**int** j = 0; j < i; j++) {  content.append((**char**)buffer[j]);  }  *//地答应http中的数据* System.***out***.println(content);  *//截取客户端要求的资源你路径demo.html，赋值给url  parseUrl*(content.toString()); } |

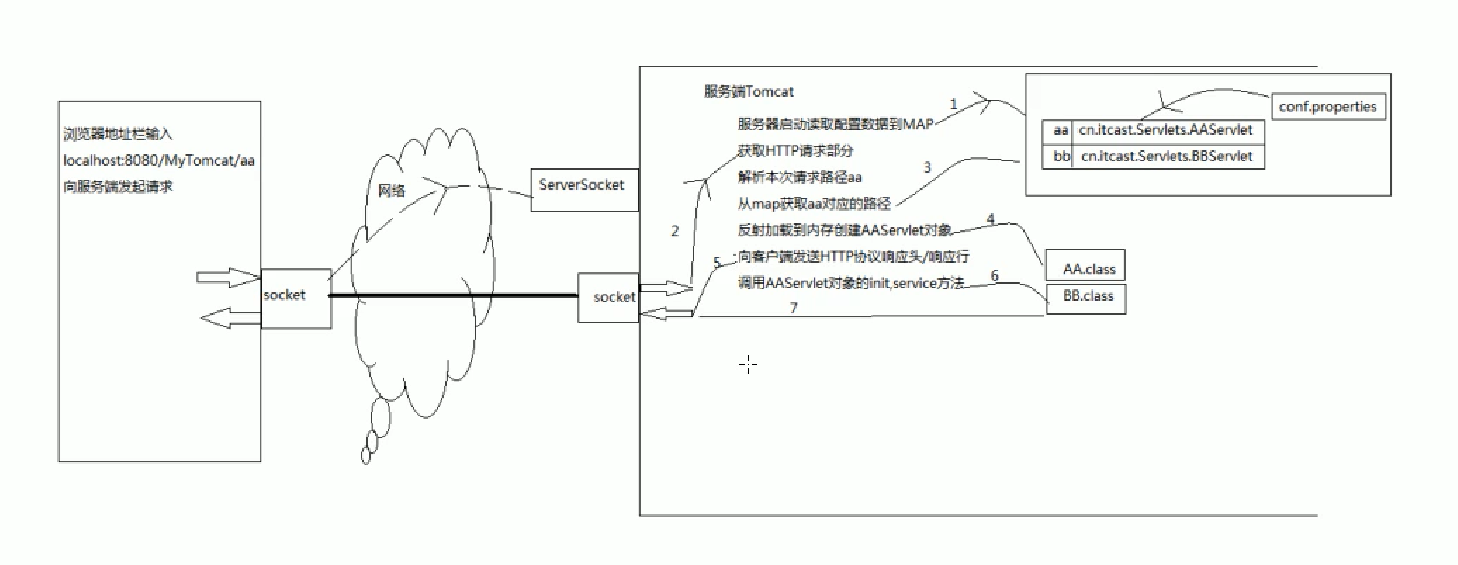
//截取客户端要求的资源你路径demo.html，赋值给url

|  |
| --- |
| *//截取客户端要求的资源你路径demo.html，赋值给url* **private static void** parseUrl(String content) {  *//定义两个变量，存放请求行的2个空格的位置* **int** index1,index2;  *//获取http请求部分的第一个空格的位置* index1 = content.indexOf(**" "**);  **if**(index1!=-1){  index2 = content.indexOf(**" "**,index1+1);  *//获取http请求部分的第2个空格的位置* **if**(index2>index1){  *url* = content.substring(index1+2,index2);  }  }  *//打印本次的url* System.***out***.println(*url*); } |

//读取访问的静态资源，并将资源返回给客户端

|  |
| --- |
| *//读取访问的静态资源，并将资源返回给客户端* **private static void** sendStaticResource(OutputStream ops) {  *//定义一个字节数组，用于存放本次请求的静态资源demo01.htm1的内容* **byte**[] bytes = **new byte**[2048];  *//定义一个文件输入流，用户获取静态资源demo01.html中的内容* FileInputStream fis = **null**;  **try** {  *// 创建文件对象File，代表本次要请求的资源demo01.html* File file = **new** File(*WEB\_Root*,*url*);  System.***out***.println(file);  *//如果文件存在* **if**(file.exists()){  *// 向客户端输出HTTP协议的响应行/响应头* ops.write(**"HTTP/1.1 200 OK\n"**.getBytes());  ops.write(**"Server:apach-Coyote/1.1\n"**.getBytes());  ops.write(**"Content-Type:text/html;charset=utf-8\n"**.getBytes());  ops.write(**"\n"**.getBytes());  *// 获取到文件输入流对象* fis = **new** FileInputStream(file);  *// 读取静态资源demo01.html中的内容到数组中* **int** ch = fis.read(bytes);  **while**(ch!=-1){  ops.write(bytes,0,ch);  ch = fis.read();  }  *// 将读取到数组中的内容通过输出流发送到客户端* }**else**{  *//如果文件不存在 向客户端响应文件不存在消息* ops.write(**"HTTP/1.1 404 not found\n"**.getBytes());  ops.write(**"Server:apach-Coyote/1.1\n"**.getBytes());  ops.write(**"\n"**.getBytes());  ops.write(**"\n"**.getBytes());  String errorMessage = **"file not found"**;  ops.write(errorMessage.getBytes());  }  }**catch** (Exception e){  e.printStackTrace();  }**finally** {  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

### 实现tomcat2



# web应用

## Web应用（重点）

静态网站：

在webapps目录下创建一个目录（命名必须不包含中文和空格），这个目录称之为项目目录；

在项目目录下创建一个html文件；//访问路径

动态网站：

在webpass目录下创建一个项目目录；

在项目目录下创建如下内容：

WEB-INF目录；

在WEB-INF目录下创建web.xml文件和一个lib文件夹

创建静态或动态页面 //访问路径：

<http://localhost:8080/hello2/index.html>

注意：动态网页的web.xml可以在webAPP目录下的root借用

### 创建静态应用

* 在webapps下创建一个hello目录；
* 在webapps\hello\下创建index.html；
* 启动tomcat；
* 打开浏览器访问http://localhost:8080/hello/index.html

index.html

|  |
| --- |
| <html>  <head>  <title>hello</title>  </head>  <body>  <h1>Hello World!</h1>  </body>  </html> |

### 创建动态应用

* 在webapps下创建hello1目录；
* 在webapps\hello1\下创建WEB-INF目录；
* 在webapps\hello1\WEB-INF\下创建web.xml；
* 在webapps\hello1\下创建index.html。
* 打开浏览器访问http://localhost:8080/hello/index.html

web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app version="2.5"  xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">  </web-app> |

完整的Web应用还需要在WEB-INF目录下创建：

* classes；
* lib目录；

webapps

|- hello

|-index.html

|-index.jsp

|-WEB-INF

|-web.xml

|-classes

|-lib

* hello：应用目录，hello就是应用的名称；
* index.html：应用资源。应用下可以有多个资源，例如css、js、html、jsp等，也可以把资源放到文件夹中，例如：hello\html\index.html，这时访问URL为：http://localhost:8080/hello/html/index.html；
* WEB-INF：这个目录名称必须是大写，这个目录下的东西是无法通过浏览器直接访问的，也就是说放到这里的东西是安全的；
* web.xml：应用程序的部署描述符文件，可以在该文件中对应用进行配置，例如配置应用的首页：

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

* classes：存放class文件的目录；
* lib：存放jar包的目录；

给MyEclipse添加tomcat容器：在servers中右击选择configure... ...,选tomcat版本，设置tomcat路径，并点击enable可显

### 配置外部应用（了解）

　　原来我们的项目放到webapps下，现在我放到外面，也希望tomcat可以找到它！

也可以把应用放到Tomcat之外，这就是外部应用了。例如我们把上面写的hello应用从webapps目录中剪切到C盘下，即C:/hello。现在hello这个Web应用已经不在Tomcat中了，这时我们需要在tomcat中配置外部应用的位置，配置的方式一共有两种：

* conf/server.xml：打开server.xml文件，找到<Host>元素，在其中添加<Context>元素，代码如下：

server.xml

|  |
| --- |
|  |

1. path：指定当前应用的名称；项目访问路径，即地址栏输入路径
2. docBase：指定应用的物理位置；项目存放路径
3. 浏览器访问路径：<http://localhost:8080/itcast_hello/index.html>。

* conf/catalana/localhost：在该目录下创建itcast\_hello.xml文件，在该文件中编写<Context>元素，代码如下：

|  |
| --- |
|  |

1. 文件名：指定当前应用的名称；网址访问与项目名称相同即文件名itcast\_hello.xml
2. docBase：指定应用的物理位置；
3. 浏览器访问路径：<http://localhost:8080/itcast_hello/index.html>。

### 理解server.xml（了解）

<Server>

<Service>

<Connector>

<Engine>

<Host>

<Context>

* <Server>：根元素，表示整个服务器的配置信息；
* <Service>：<Server>的子元素，在<Server>中只能有一个<Service>元素，它表示服务；
* <Connector>：<Service>的子元素，在<Service>中可以有N个<Connector>元素，它表示连接。
* <Engine>：<Service>的子元素，在<Service>中只能有一<Engine>元素，该元素表示引擎，它是<Service>组件的核心。
* <Host>：<Engine>的子元素，在<Engine>中可以有N个<Host>元素，每个<Host>元素表示一个虚拟主机。所谓虚拟主机就像是真的主机一样，每个主机都有自己的主机名和项目目录。例如<Host name=”localhost” appBase=”webapps”>表示主机名为localhost，这个主机的项目存放在webapps目录中。访问这个项目下的主机时，需要使用localhost主机名，项目都存放在webapps目录下。
* <Context>：<Host>元素的子元素，在<Host>中可以有N个<Context>元素，每个<Context>元素表示一个应用。如果应用在<Host>的appBase指定的目录下，那么可以不配置<Context>元素，如果是外部应用，那么就必须配置<Context>。如果要为应用指定资源，也需要配置<Context>元素。

我们可以把<Server>看作是一个大酒店：

* <Service>：酒店的服务部门；
* <Connector>：服务员；
* <Engine>：后厨；
* <Host>：后厨中的一个区，例如川菜区是一个<Host>、粤菜区是一个<Host>；
* <Context>：后厨的一个厨师。

用户发出一个请求：http://localhost:8080/hello/index.jsp。发现是http/1.1协议，而且还是8080端口，所以就交给了处理这一请求的“服务员（处理HTTP请求的<Connector>）”，“服务员”再把请求交给了“后厨（<Engine>）”，因为请求是要一盘水煮鱼，所以由“川菜区（<Host>）”负责，因为“大老王师傅<Context>”做水煮鱼最地道，所以由它完成。

* <Connector>：关心请求中的http、和8080；
* < Host>：关心localhost；
* <Context>：关心hello

### 映射虚拟主机（了解）

我们的目标是，在浏览器中输出：<http://www.itcast.cn>就可以访问我们的项目。

完成这一目标，我们需要做三件事：

* 修改端口号为80，这一点应该没有问题吧；
* 在本机上可以解析域名为127.0.0.1，这需要修改C:\WINDOWS\system32\drivers\etc\hosts文件，添加对<http://www.itcast.cn>和127.0.01的绑定关系；
* 在server.xml文件中添加一个<Host>（主机）。

1. 修改端口号为80



1. 绑定<http://www.itcast.cn>和127.0.01的绑定关系



1. server.xml文件中添加一个<Host>



* name=”www.itcast.cn”：指定虚拟主机名为[www.itcast.cn](http://www.itcast.cn)；
* appBase=”F:/itcastapps”：指定当前虚拟主机的应用程序存放目录为F:/itcastapps。
* 在itcastapps目录下创建名为ROOT的应用，因为一个主机只可以有一个名为ROOT的应用，名为ROOT的应用在浏览器中访问是可以不给出应用名称。



现在访问：<http://www.itcast.cn>看看是什么页面！

请注意，只有本机可以通过<http://www.itcast.cn>来访问，而其他电脑不可以！

### MyEclipse创建JavaWeb应用

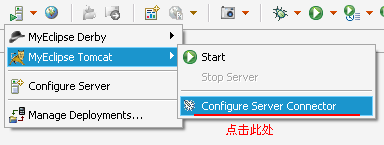
## 配置Tomcat

使用MyEclipse配置服务器后，就可以使用MyEclipse来启动和停止服务器了。当然，你需要先安装好服务器（Tomcat），才能配置。

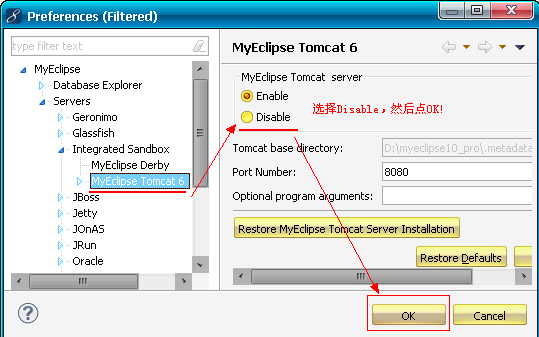
MyEclipse自带了一个Tomcat，强烈建议不要使用它。所以，我们需要先把MyEclipse自带的Tomcat关闭，然后再来配置我们自己的Tomcat。

* 关闭MyEclipse自带Tomcat。

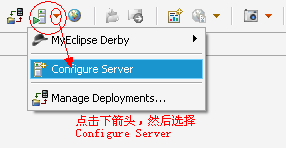
在工具栏中找到，点击下箭头，点击Configure Server Connector。

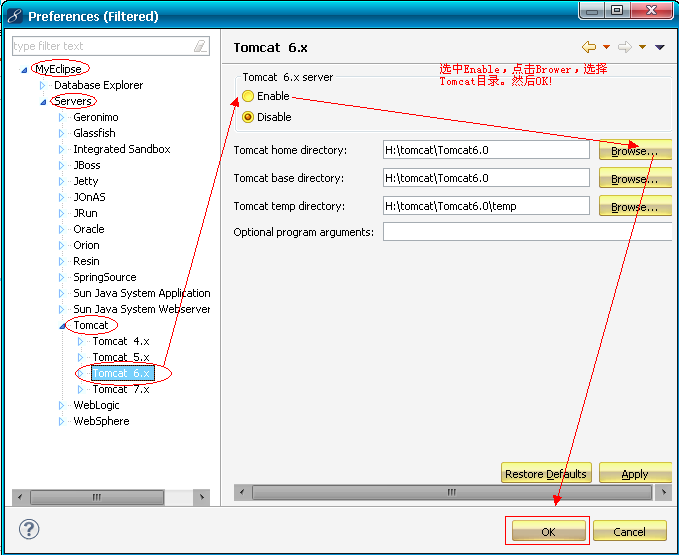


弹出对话框



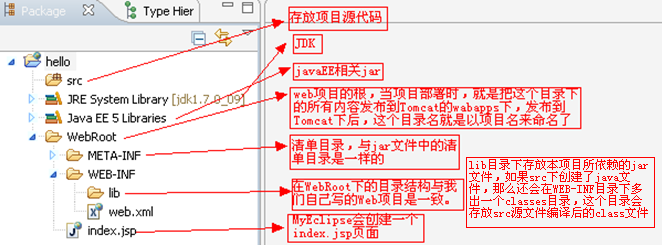
* 配置我们自己的Tomcat





使用MyEclipse启动Tomcat

### 创建JavaWeb应用



### 启动Tomcat

　　上面两种方式都可以启动tomcat

### 关闭tomcat

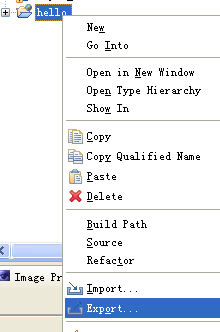
　　与启动Tomcat相同位置下方就是Stop Server，即可关闭tomcat了。

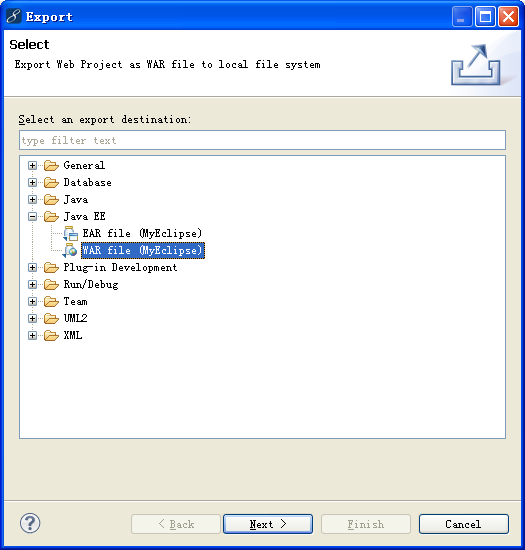
### 发布项目到tomcat的webapps目录

　　项目发布后，就是把项目的WebRoot目录copy到Tomcat的webapps目录，并把WebRoot重命名为项目名称，即hello。所以在Tomcat的webapps下会多出一个文件夹hello。

### 打war包

JavaSE程序可以打包成Jar包，而JavaWeb程序可以打包成war包。然后把war发布到Tomcat的webapps目录下，Tomcat会在启动时自动解压war包。





# Servlet

## 什么是servlet（Server + let运行在服务器端的小程序）

Servlet是JavaWeb的三大组件之一，它属于动态资源。Servlet的作用是处理请求，服务器会把接收到的请求交给Servlet来处理，在Servlet中通常需要：

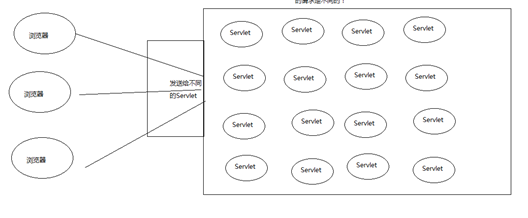
接收请求数据；

 处理请求；

 完成响应；

例如客户端发出登录请求，或者输出注册请求，这些请求都应该由Servlet来完成处理！Servlet需要我们自己来编写，每个Servlet必须实现javax.servlet.Servlet接口。

狭义的Servlet是指Java语言实现的一个接口，广义的Servet是指任何实现了这个Servlet的类，一般情况下，人们将Servlet理解为后者。



Servelet 有多个 tomcat 来将请求引导到不同的servlet。

## servlet和jsp的区别

### Servlet是Java代码，JSP是页面代码

编写Servlet就是编写Java代码，所以应用Java中的规范去编写Servlet类就可以了，但是若想在客户端中响应结果，就必须在代码中加入大量的HTML代码，可想而知当想要得到一个比较美观、复杂的界面时，HTML代码量会相当多而且繁琐。JSP是以HTML代码为主，在页面中适当嵌入Java代码来处理业务上的逻辑。显然，JSP会比Servlet较易编写和美观。基于这点的差异，也是选择Servlet或者JSP技术的考量标准之一。如果业务中主要是以页面为主，则选择JSP技术；反之，则选择Servlet技术，它适合服务器端开发。

### Servlet的运行速度快过JSP

因为Servlet本身就是一个Java类，编译的时候直接被转换为class类文件。而JSP需要先被编译为Java类，而后再运行，所以Servlet的运行速度较快。

### 编辑Html工具支持编辑Servlet

了解了Servlet与JSP的差别之后，在编写Web应用程序时，就要根据当前的需要权衡使用JSP或者Servlet。开发者应尽量使JSP和Servlet都发挥出最大的作用，又能够便于日后的代码维护工作。一般而言，Servlet大多用于负责对客户端的请求进行处理和调用Java Bean，由Java Bean负责提供可复用的数据以及数据的访问数据等；而JSP页面主要负责页面的展示，将动态数据展现给客户，这就是开发者提出的简易MVC模式，这样分工大大减少了JSP页面中Java程序和HTML代码的耦合度，对维护工作具有重大的意义。

## 主要功能

1. 读取客户端（浏览器）发送的显式的数据。这包括网页上的 HTML 表单。

2. 读取客户端（浏览器）发送的隐式的 HTTP 请求数据。这包括cookies、媒体类型和浏览器能理解的压缩格式等等。

3. 处理数据并生成结果。这个过程可能需要访问数据库。

4. 发送显式的数据（即文档）到客户端（浏览器）。该文档的格式可以是多种多样的，包括文本文件（HTML 或 XML）、二进制文件（GIF 图像）、Excel 等。

5. 发送隐式的 HTTP 响应到客户端（浏览器）。这包括告诉浏览器或被返回的文档类型（例如 HTML），设置 cookies 和缓存参数，以及其他类似的任务。

1、对客户端发送的数据进行读取和拦截

客户端在发送一个请求时，一般而言都会携带一些数据（例如：URL中的参数、页面中的表单、Ajax提交的参数等），当一个Servlet接收到这些请求时，Java Servlet中的类通过所提供的方法就能得到这些参数（例如：方法 request.getParameterName(name)用于获得名为name的参数值），也正因为这个原因，Servlet可以对发送请求起到拦截作用，它在某些请求前先做一个预处理分析，从而判断客户端是否可以做某些请求（例如：检查访问权限、设定程序的字符集、检查用户角色等），当Servlet具有如上功能时，一般称之为拦截器。

2.读取客户端请求的隐含数据

客户端请求的数据可以分为隐含数据和显式数据：隐含数据一般是不直接跟随于URL中，它存在于请求的来源、缓存数据（Cookie)、客户端类型中；显式数据显然是用户可以直观看到的（例如表单数据和URL参数）。Servlet不但可以处理显式数据，而且可以处理隐含数据，是个“多面手”。

3.运行结果或者生成结果

当一个Web应用程序对客户端发出的请求做出响应时，一般需要很多中间过程才能得到结果，Servlet 就是起到这个中间角色的功能，它协调各组件、各部分完成相应的功能，根据不同的请求做出相应的响应并显示结果。

4.发送响应的数据

Servlet在对客户端做出响应并经过处理得出结果后，会对客户端发送响应的数据，以便让客户端获取请求的结果数据。在Web应用程序中，Servlet的这个作用相当突出，无论现有的技术多么突出，都是基于这个作用点出发的。

## web容器的作用

Web容器的作用就是创建一个Servlet实例，并完成Servlet注册以及根据web.xml中的URL进行响应。当请求来到容器时，Web容器会转发给对应的Servlet来处理请求。

当客户端请求HTTP服务器时，会使用HTTP来传递请求、标头、参数等信息。HTTP协议是无意识的协议，通过文本信息传递消息，而Servlet是Java对象，运行在Web容器中。当HTTP服务器将请求转给Web容器时，Web容器会创建一个HttpServletRequest和HttpServletResponse对象，将请求中的信息传递给HtpServletRequest对象，而HttpServletResponse对象则作为对客户端响应的Jaya对象。

Web容器会根据配置信息（例如web.xml或者@WebServlet)，查找相对应的Servlet 并调用它的service()方法。service0方法会根据HTTP请求的方式，决定是调用doPost()方法或者是doGet()方法。例如，HTTP请求的方式为post，则调用doPost0方法。

## 实现Servlet的方式

Servet是一个接口，通常一般会用HttpServlet继承Servlet接口，服务会从Servlet实现对象中去寻找接口中的方法去调用init()、service()、destory()方法。

Servlet接口一个规则，规定如何去编写代码，按照这个接口来写，服务器能够识别，这就解决了服务器如何处理java代码。

 实现javax.servlet.Servlet接口；

 继承javax.servlet.GenericServlet类；实现Servlet接口的类。

 继承javax.servlet.http.HttpServlet类；//通用方式，最简单与http协议相关。

Servlet类由我们来写，但对象由服务器来创建，并且由服务器来调用相应的方法。

通常我们会去继承HttpServlet类来完成我们的Servlet，但学习Servlet还要从javax.servlet.Servlet接口开始学习。

Servlet的接口代码

|  |
| --- |
| **public** **interface** Servlet {  **public** **void** init(ServletConfig config) **throws** ServletException;  **public** ServletConfig getServletConfig();  **public** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException;  **public** String getServletInfo();  **public** **void** destroy();  } |

## 访问Servlet的执行过程

### 浏览器地址请求

1.http://localhost:80/servlet/hello

Localhost -> 找到对应的主机地址。

80->找到对应的服务器端口号。

URI：虚拟项目名/servlet别名

2.到tomcat的server.xml中找到 Context 这个配置

<Context docBase="E:\java\javaee\day13-servlet\webapps" path="servlet" />

3.  servlet: 这个Servlet找到Context中的path

4.  通过path找到它对应的docBase,也就是找到我们工程实际地址

5.  Hello 就到我们实现的项目中找到相应的

      web.xml文件中的servlet-mapping中的url-pattern

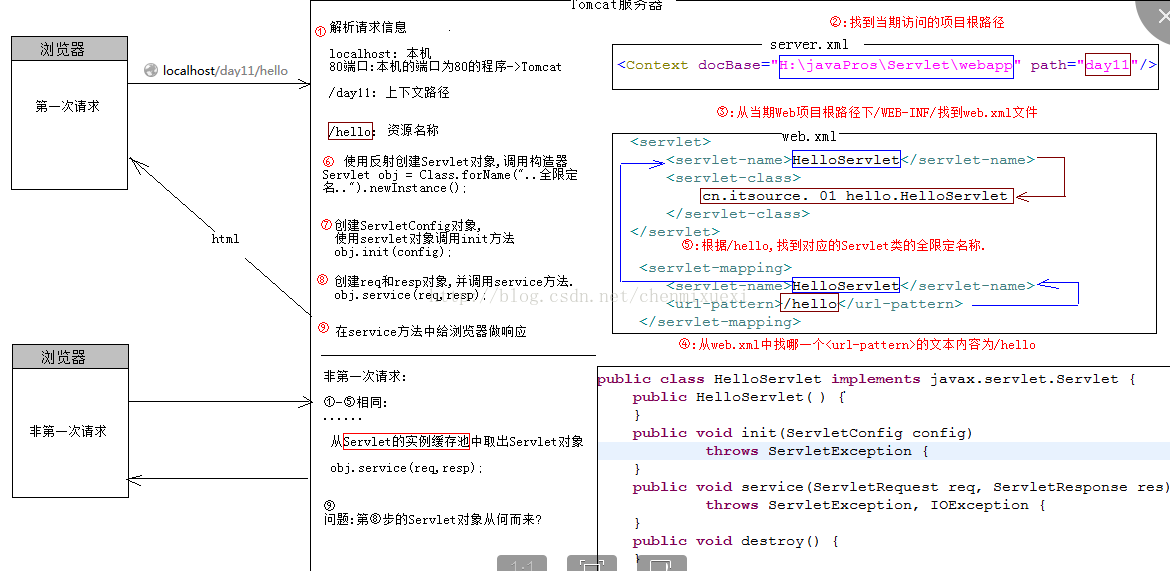
### web.xml中

1 先找到url-pattern –> <url-pattern>/hello</url-pattern>。

2. 找到它对应的servlet-name。

3 .通过mapping中servlet-name找到相应的servlet(它们的servet-name是一样的)在servlet标签中找到它的servlet-class，它里面是全限定名称.

|  |
| --- |
| <servlet>  <servlet-name>hello</servlet-name>  <servlet-class>cn.itcast.servlet.HelloServlet</servlet-class>  </servlet> 通过反射创建该对象。  <servlet-mapping>  http://localhost:8080/day09/Aservlet  <servlet-name>hello</servlet-name>  <url-pattern>/helloworld</url-pattern>  </servlet-mapping> |



## 配置XML

目的：Servlet程序必须在WEB应用程序的web.xml文件中进行注册和映射其访问路径，才可以被Servlet引擎加载和被外界访问。

服务器找到Webapp下的项目，项目中可能有多个不同的功能的servlet，通过输入的地址用xml匹配servlet。调用相应的方法，实现功能。

一个<servlet>元素用于注册一个Servlet，它包含有两个主要的子元素：<servlet-name>和<servlet-class>，分别用于设置Servlet的注册名称中和Servlet的完整类名。

一个<servlet-mapping>元素用于映射一个已注册的Servlet的一个对外访问路径，它包含有两个子元素：<servlet-name>和<url-pattern>，分别用于指定Servlet的注册名称和Servlet的对外访问路径。

< servlet >

<servlet-name> hello </ servlet-name>注册名

<servlet-class > cn.itcast.servlet.HelloServlet</servlet-class>servlet全限定类名

<load-on-startup> n</load-on-startup>

</ servlet>

< servlet-mapping>

<servlet-name> hello </ servlet-name>注册名

<url-pattern>/helloworld</url-pattern>

servlet对外访问路径(映射路径)，路径以斜杠开头。

<url-pattern>/AServlet </url-pattern >

<url-pattern>/BServlet </url-pattern >

</ servlet-mapping>

### 配置绑定：

目的在 web.xml 中配置Servlet 的目的其实只有一个，就是把访问路径与一个Servlet 绑定到一起，上面配置是把访问路径：“ /helloworld ”与“ cn.itcast.servlet.HelloServlet ”绑定到一起。

<servlet> ：指定HelloServlet这个Servlet 的名称为hello；

<servlet-mapping> ：指定 /helloworld访问路径所以访问的Servlet 名为 hello 。

<servlet> 和<servlet-mapping> 通过 <servlet-name> 这个元素关联在一起。

注：1、同一个Servlet可以被映射到多个URL上，即多个<servlet-mapping>元素的<serlet-name>子元素的设置值可以是同一个Servlet的注册名。

<url-pattern>n<url-pattern>:<url-pattern> 是<servlet-mapping>的子元素，用来指定 Servlet的访问路径 ，即URL。它必须是以 “ / ”开头！

2、可以在<servlet-mapping>中给出多个<url-pattern>，Servlet 绑定了三个 URL，无论访问 /helloworld 、/AServlet 还是 /BServlet ，访问的都是cn.itcast.servlet.HelloServlet。

3、通配符“\*” 匹配任何URL前缀或后缀

第一种格式：已斜杠开头，并已/\*结尾。

例如：<url-pattern>/servlet/\*<url-patter>

/servlet/a 、 /servlet/b ，都匹配 /servlet/\* ；路径匹配

<url-pattern>\*.do</url-pattern>

/abc/def/ghi.do、/a.do ，都匹配 \*.do ；扩展名匹配

第二种格式：\*.扩展名。

例如：<url-pattern>/\*<url-pattern> 匹配所有URL；

匹配原则：优先级，匹配的越少，优先级越高

请注意，通配符要么为前缀，要么为后缀，不能出现在URL 中间位置，也不能只有通配符。例 如： /\*.do就是错误的，因为星号出现在URL 的中间位置上了。\*.\* 也是不对的，因为一个URL 中最多只能出现一个通配符。

通配符是一种模糊匹配URL的方式，如果存在更具体的<url-pattern> ，那么访问路径会去匹配具体的 <url-pattern>。

### 配置启动优先级

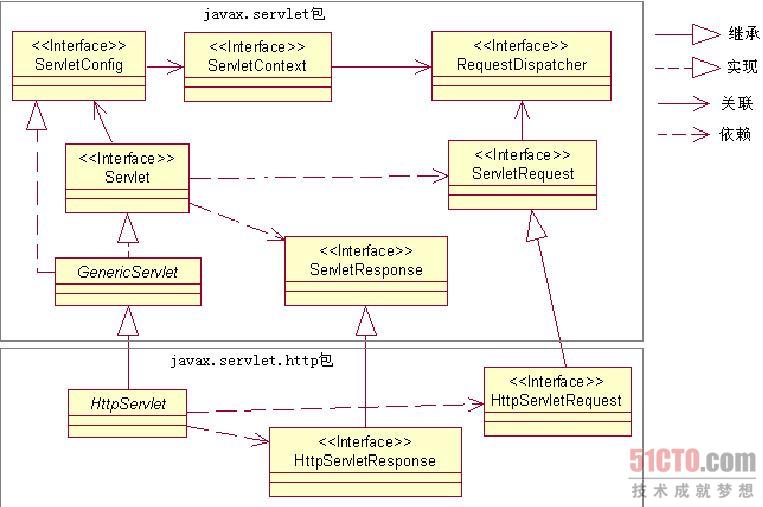
<load-on-startup>n<load-on-startup>

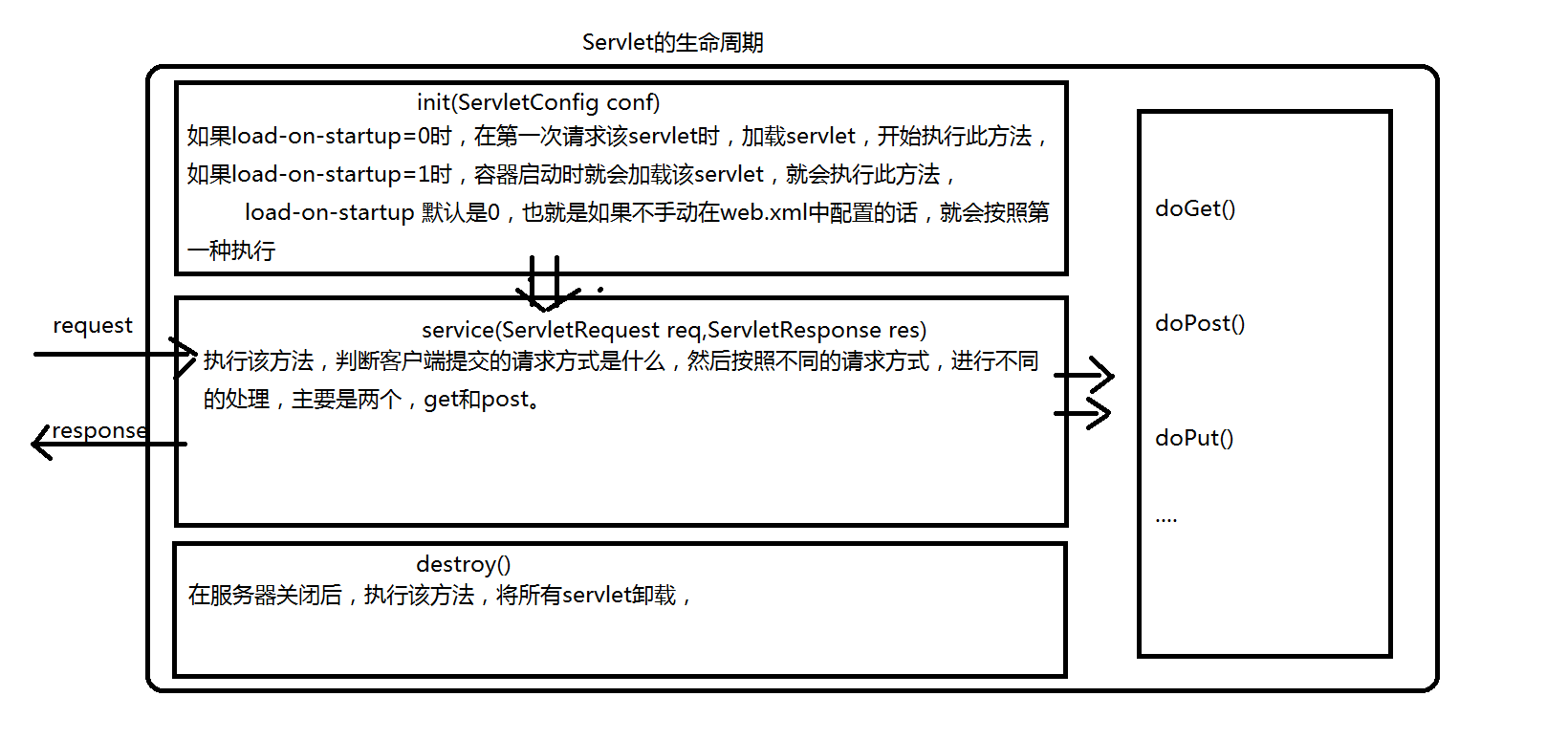
在 <servlet> 元素中配置 <load-on-startup> 元素可以让服务器在启动时就创建该Servlet ，其中<load-on-startup> 元素的值必须是大于等于的整数，它的使用是服务器启动时创建Servlet 的顺序。

N为负数：则在第一次请求时被创建。

N为0或者整数：则哎当前web应用被servlet容器加载时创建实例，且数越小越早创建。

## servlet接口





### servlet生命周期方法（以下方法由servlet容器负责调用）

所谓xxx的生命周期，就是说xxx的出生、服务，以及死亡。Servlet生命周期也是如此！与Servlet的生命周期相关的方法有：

构造器：只被调用一次，只有第一次请求Servlet时，创建Servlet，调用构造器。标明servlet为单实例，可以在这里可以进行初始化设置。

void init(ServletConfig) ；只被调用一次，在创建好实例后立即被调用，用于初始化当前的servlet。

void service(ServletRequest , ServletResponse) ；被多次调用，每次请求都会调用service方法，实际用于响应请求，该方法只有在有请求访问的时候才会被调用。

void destroy()；只被调用一次，在当前servlet所造的web应用卸载前被调用，用于释放当前servlet所占用的资源，不确定用户什么时候停止访问，所以只有服务器关闭的时候，servlet才会被关闭。

### servlet的出生

服务器会在Servlet第一次被访问时创建Servlet，或者是在服务器启动时创建Servlet。如果服务器启动时就创建Servlet，那么还需要在web.xml文件中配置。也就是说默认情况下，Servlet是在第一次被访问时由服务器创建的。

而且一个Servlet类型，服务器只创建一个实例对象，例如在我们首次访问http://localhost:8080/helloservlet/helloworld时，服务器通过“/helloworld”找到了绑定的Servlet名称为cn.itcast.servlet.HelloServlet，然后服务器查看这个类型的Servlet是否已经创建过，如果没有创建过，那么服务器才会通过反射来创建HelloServlet的实例。当我们再次访问http://localhost:8080/helloservlet/helloworld时，服务器就不会再次创建HelloServlet实例了，而是直接使用上次创建的实例。

在Servlet被创建后，服务器会马上调用Servlet的void init(ServletConfig)方法。请记住， Servlet出生后马上就会调用init()方法，而且一个Servlet的一生。这个方法只会被调用一次。这好比小孩子出生后马上就要去剪脐带一样，而且剪脐带一生只有一次。

我们可以把一些对Servlet的初始化工作放到init方法中！

### servlet服务

　当服务器每次接收到请求时，都会去调用Servlet的service()方法来处理请求。服务器接收到一次请求，就会调用service() 方法一次，所以service()方法是会被调用多次的。正因为如此，所以我们才需要把处理请求的代码写到service()方法中！

service的处理流程：

1.设置请求编码格式

2.设置响应编码格式

2.获取请求数据

3.处理请求数据

4.响应处理结果

### servlet的离去

Servlet是不会轻易离去的，通常都是在服务器关闭时Servlet才会离去！在服务器被关闭时，服务器会去销毁Servlet，在销毁Servlet之前服务器会先去调用Servlet的destroy()方法，我们可以把Servlet的临终遗言放到destroy()方法中，例如对某些资源的释放等代码放到destroy()方法中。destroy()没有销毁servlet的功能，之时servlet销毁前调用该方法。

### 测试声明周期方法

|  |
| --- |
| **public** **class** HelloServlet **implements** Servlet {  public HelloServlet(){  System.*out*.println("实例化！");  }  **public** **void** init(ServletConfig config) **throws** ServletException {  System.*out*.println("Servlet被创建了！");  }  **public** ServletConfig getServletConfig() {**return** **null**;}  **public** **void** destroy() {  System.*out*.println("Servlet要离去了！");  }  **public** String getServletInfo() {**return** **null**;}关于servlet的相关信息，一般没什么用  **public** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException {  System.*out*.println("hello servlet!");  }  } |

在首次访问HelloServlet时，init方法会被执行，而且也会执行service方法。再次访问时，只会执行service方法，不再执行init方法。在关闭Tomcat时会调用destroy方法。

## Servlet接口相关的类型

在Servlet接口中还存在三个我们不熟悉的类型：

 1.ServletRequest：service() 方法的参数，它表示请求对象，它封装了所有与请求相关的数据，它是由服务器创建的；

 2.ServletResponse：service()方法的参数，它表示响应对象，在service()方法中完成对客户端的响应需要使用这个对象；比如发状态码。

 3.ServletConfig：init()方法的参数，它表示Servlet配置对象，它对应Servlet的配置信息，那对应web.xml文件中的<servlet>元素。

### ServletRequest和ServletResponse

ServletRequest和ServletResponse是Servlet#service() 方法的两个参数，一个是请求对象，一个是响应对象，可以从ServletRequest对象中获取请求数据，可以使用ServletResponse对象完成响应。你以后会发现，这两个对象就像是一对恩爱的夫妻，永远不分离，总是成对出现。

ServletRequest和ServletResponse的实例由服务器创建，然后传递给service()方法。如果在service() 方法中希望使用HTTP相关的功能，那么可以把ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse。这也说明我们经常需要在service()方法中对ServletRequest和ServletResponse进行强转，这是很心烦的事情。不过后面会有一个类来帮我们解决这一问题的。

HttpServletRequest方法：

String getParameter(String paramName)：获取指定请求参数的值；

 String getMethod()：获取请求方法，例如GET或POST；

 String getHeader(String name)：获取指定请求头的值；

 void setCharacterEncoding(String encoding)：设置请求体的编码！因为GET请求没有请求体，所以这个方法只对POST请求有效。当调用request.setCharacterEncoding(“utf-8”)之后，再通过getParameter()方法获取参数值时，那么参数值都已经通过了转码，即转换成了UTF-8编码。所以，这个方法必须在调用getParameter()方法之前调用！

HttpServletResponse方法

PrintWriter getWriter()：获取字符响应流，使用该流可以向客户端输出响应信息。例如response.getWriter().print(“<h1>Hello JavaWeb!</h1>”)；

 ServletOutputStream getOutputStream()：获取字节响应流，当需要向客户端响应字节数据时，需要使用这个流，例如要向客户端响应图片；

 void setCharacterEncoding(String encoding)：用来设置字符响应流的编码，例如在调用setCharacterEncoding(“utf-8”);之后，再response.getWriter()获取字符响应流对象，这时的响应流的编码为utf-8，使用response.getWriter()输出的中文都会转换成utf-8编码后发送给客户端；

 void setHeader(String name, String value)：向客户端添加响应头信息，例如setHeader(“Refresh”, “3;url=http://www.itcast.cn”)，表示3秒后自动刷新到http://www.itcast.cn；

 void setContentType(String contentType)：该方法是setHeader(“content-type”, “xxx”)的简便方法，即用来添加名为content-type响应头的方法。content-type响应头用来设置响应数据的MIME类型，例如要向客户端响应jpg的图片，那么可以setContentType(“image/jepg”)，如果响应数据为文本类型，那么还要台同时设置编码，例如setContentType(“text/html;chartset=utf-8”)表示响应数据类型为文本类型中的html类型，并且该方法会调用setCharacterEncoding(“utf-8”)方法；

 void sendError(int code, String errorMsg)：向客户端发送状态码，以及错误消息。例如给客户端发送404：response(404, “您要查找的资源不存在！”)。

4、响应请求的过程

(1)Servlet引擎检查是已经装载并创建了该Serviet的实例对象。如果是，则直接执行第（4）步，否则，执行第(2)步。

(2)装载并创建该Servlet的一个实例对象：调用该Servlet的构造器.

(3)调用Servlet实例对象的init()方法。

(4)创建一个用于封装请求的SerlvetRequest对象和一个代表响应消息的ServletResponse对象，然后调用Servlet的service()方法并将请求和响应对象作为参数传递进去。

(5)WEB应用程序被停止或重新启动之前，Servlet引擎将卸载Servlet，并在卸载之前调用Servlet的destroy()方法。

## ServletConfig接口

当容器初始化Servlet时，会为Servlet创建唯一的ServletConfig对象。利用Web容器读取web.xml文件，将初始化参数传给ServletConfig，而ServletConfig作为对象参数传递到init()方法中。ServletConfig对象封装了Servlet对象的配置信息。也就是说ServletConfig对象包含了Servlet类在web.xml文件的<servlet>标签中配置的信息。

### ServletConfig接口中的方法

String getServletName() ：获取 Servlet 在 web.xml 文件中的配置名称，即<servlet-name> 指定的名称；//获取servlet的名称，也就是我们在web.xml中配置的servlet-name。

|  |
| --- |
| String s = this.getServletConfig().getServletName();  System.out.println(s);//这里返回 |

ServletContext getServletContext() ：用来获取ServletContext对象，ServletContext 会在后面讲解；（重点）

String getInitParameter(String name) ：用来获取在web.xml 中配置的初始化参数，通过参数名来获取参数值；获取在servlet中初始化参数的值。这里注意与全局初始化参数的区分。这个获取的只是在该servlet下的初始化参数

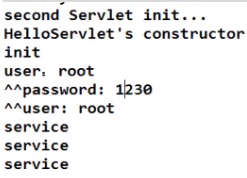
Enumeration getInitParameterNames() ：用来获取在 web.xml 中配置的所有初始化参数名称。获取在Servlet中所有初始化参数的名字，也就是key值，可以通过key值，来找到各个初始化参数的value值。注意返回的是枚举类型

1、配置初始化参数：(该参数在<servlet>标签里面，是servlet的初始化参数)

|  |
| --- |
| <servlet>  <servlet-name> hello </ servlet-name>  <servlet-class > cn.itcast.servlet.HelloServlet</servlet-class>  <l--配置 Serlvet的初始化参数。且节点必须在1oad-on-startup节点的前面-->  <init-param>  <！--参数名-->  <param-name>user</param-name>  <！--参数值-->  <param-value>root</param-value>  </init-param>  <init-param>  <param-name>password</param-name>  <param-value>1230</param-value>  </init-param>  <1oad-on-startup>-1</1oad-on-startup>  </servlet> |

(2)获取初始化参数：

|  |
| --- |
| 1、获取单个初始化参数  String getInitParameter(String name)  string user=servletConfig.getInitParameter("user")；  System.out.println(user)；//返回root  2、如果有多个初始化参数  Enumeration getInitParameterNames()  Enumeration<String>names=servletConfig.getInitParameterNames()；  while(names.hasMoreElements()){  String name=names.nextElement()；  String value = servletConfig.getInitParameter(name);  System.out.println("name:"+name+“value”+value)；  }  结果为：password：1230  user：root |



### ServletConfig 对象及其使用

1、补充：Context和ContextPath ： 一个Web工程，如名为JavaWeb，访问的路径为http://localhost:8080/JavaWeb，这整个web应用就成为一个Context，路径/JavaWeb被称为上下文路径(ContextPath )。request.getContextPath ()可以获取。

2、ServletConfig： 当servlet配置了初始化参数后，web容器在创建servlet实例对象时，会自动将这些初始化参数封装到ServletConfig对象中，并在调用servlet的init()方法时，将ServletConfig 对象传递给servlet。进而，通过ServletConfig对象就可以得到当前servlet的初始化参数。

### 应用

1）获得字符集编码

String charset = this.config.getInitParameter("charset");

2）获得数据库连接信息

String url = this.config.getInitParameter("url");

String username = this.config.getInitParameter("username");

String password = this.config.getInitParameter("password");

3）获得配置文件

String configFile = this.config.getInitParameter("config");

## ServletRequest和ServletResponse

服务器接收到浏览器的请求后，会创建一个Request对象，对象中存储了此次请求的相关请求数据。服务器子啊调用Servlet时会将创建的Request对象作为实参传递给Servlet的方法，比如：service方法。

Public void service(ServletRequest,ServletResponse)

ServletRequest:封装了请求信息。可以从中获取到任何的请求信息。

ServletResponse:封装了响应信息，如果想给用户什么响应，具体可以使用该接口的方法实现。

### ServletRequest接口（查看api中的方法）

request 是Servlet.service()方法的一个参数，类型为javax.servlet.http.HttpServletRequest。在客户端发出每个请求时，服务器都会创建一个request 对象，并把请求数据封装到 request 中，然后在调用 Servlet.service() 方法时传递给 service()方法，这说明在service()方法中可以通过 request 对象来获取请求数据。

1、获取请求参数（servletrequest封装了请求信息）

String getParameter(String name):根据请求参数的名字，返回参数值。请求参数有多个值（例如 checkbox），该方法只能获取到第一个提交的值。

String[]getParameterValues(String name):根据请求参数的名字，返回请求参数对应的字符串数组。

Enumeration getParameterNames():返回参数名对应的Enumeration对象，似于ServletConfig(或ServletContext)的getInitParameterNames()方法。

例如：

Enumeration<String> names = request.getParameterNames();

while（names.hasMoreElements）(){

String name=names.nextElement()；

String val=request.getParameter(name)；

system.out.print1n(name+":"+val)；

}

Map getParameterMap():返回请求参数的键值对：key：参数名，value：参数值，String 数组类型。

例如：

Map<String,String[]>map=request.getParameterMap();

for(Map.Entry<String,String[]>entry:map.entrySet()){

System.out.printin(entry.getkey()+\":\"+Arrays.aslist(entry.getValue()))；

2、获取请求的URL(与http协议相关的要转换为httpSevletRequest)

HttpServletRequest httpServletRequest=(HttpServletRequest)request；

string requestURI=httpServletRequest.getRequestURI()；

System.out.println(requestURI)；

3、获取请求方式

String method=httpServletRequest.getMethod()；

System.out.print1n(method)；

4、若是一个GET请求，获取请求参数对应的那个字符串，即？后的那个字符串。

string queryString=httpServletRequest.getQueryString();

System.out.println(queryString)；

注：如果是POST，返回值为null。

5、和attribute相关的几个方法

### 3、ServletResponse（接口方法）

response是Servlet.service 方法的一个参数，类型为 javax.servlet.http.HttpServletResponse。在客户端发出每个请求时，服务器都会创建一个response 对象并传入给 Servlet.service()方法。response 对象是用来对客户端进行响应的，这说明在service()方法中使用response 对象可以完成对客户端的响应工作。

1、响应正文：response是响应对象，向客户端输出响应正文响应体可以使用 response的响应流，repsonse一共提供了两个响应流对象。

getWriter():返回Printwriter对象。调用该对象的print()方法，将把print()中的参数直接打印到客户的浏览器上。

PrintWriter out = response.getWriter() ：获取字符流

ServletOutputStream out = response.getOutputStream()：获取字节流 ；

注：在一个请求中，不能同时使用这两个流！也就是说，要么你使用repsonse.getWriter() ，要么使用 response.getOutputStream() ，但不能同时使用这两个流。不然会抛出IllegalStateException异常。

2、设置响应内容类型 ：

设置响应的内容类型：response.setContentType("application/msword")；

例如：返回一个word文档

response.setContentType("application/msword")；//word的格式

PrintWriter out=response.geturiter()；

out.print1n("helloworld...")；//返回一个word，word的内容是“helloworld”

## GenericServlet（Servlet的实现类、抽象类）

GenericServlet是Servlet接口的实现类，我们可以通过继承GenericServlet来编写自己的Servlet。下面是GenericServlet类的源代码：

GenericServlet.java 源码：

|  |
| --- |
| **public** **abstract** **class** GenericServlet **implements** Servlet, ServletConfig,  java.io.Serializable {  **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** **transient** ServletConfig config;  **public** GenericServlet() {}  @Override  **public** **void** destroy() {}  @Override  **public** String getInitParameter(String name) {  **return** getServletConfig().getInitParameter(name);  }  @Override  **public** Enumeration<String> getInitParameterNames() {  **return** getServletConfig().getInitParameterNames();  }  @Override  **public** ServletConfig getServletConfig() {  **return** config;  }  @Override  **public** ServletContext getServletContext() {  **return** getServletConfig().getServletContext();  }  @Override  **public** String getServletInfo() {  **return** "";  }  @Override  **public** **void** init(ServletConfig config) **throws** ServletException {  **this**.config = config; （如果没有添加）  **this**.init();//调用servlet的init方法，进行初始化。  }  //继承该类的类，在init中添加初始化参数时，如果重写该方法，会可能覆盖其中的参数，导致connfig为空，这里给他添加一个初始化方法，如果需要重写时，只需要重写下面的方法，服务器调用方法时会自动的调用下面的方法。  **public** **void** init() **th*rows*** *ServletException {}*  **public** **void** log(String msg) {  getServletContext().log(getServletName() + ": " + msg);  }  **public** **void** log(String message, Throwable t) {//  getServletContext().log(getServletName() + ": " + message, t);  }  @Override  **public** **abstract** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException;  @Override  **public** String getServletName() {  **return** config.getServletName();  }  } |

1. GenericServlet是Servlet接口和ServletConfig接口的实现类，但是一个抽象类。其中的service方法为抽象方法。
2. 如果新建的servlet程序直接继承GenericServlet开发会更加简便。
3. 在GenericServlet中声明的ServletCionfig类型的成员变量，在init（ServletConfig）方法中对其进行初始化。
4. 还定义了一个init()方法，在init（Serlvetconfig)方法中对其进行调用，子类可以直接覆盖init()在其中实现对Servlet的初始化。
5. 不建议直接覆盖init（Servletconfig），因为如果忘记编写 super（Servletconfig)，而还是用了Serlvetconfig接口的方法，则会出现空指针异常。（第二种解决方法）
6. 新建的init()并非servlet的生命周期方法，而init（ServletConfig）是生命周期的相关方法。

### GenericServlet的init()方法

在GenericServlet中，定义了一个ServletConfig config实例变量，并在init(ServletConfig)方法中把参数ServletConfig赋给了实例变量。然后在该类的很多方法中使用了实例变量config。

如果子类覆盖了GenericServlet的init(StringConfig)方法，那么this.config=config这一条语句就会被覆盖了，也就是说GenericServlet的实例变量config的值为null，那么所有依赖config的方法都不能使用了。如果真的希望完成一些初始化操作，那么去覆盖GenericServlet提供的init()方法，它是没有参数的init()方法，它会在init(ServletConfig)方法中被调用。

### 实现了ServletConfig接口

GenericServlet还实现了ServletConfig接口，所以可以直接调用getInitParameter()、getServletContext()等ServletConfig的方法。

## **HttpServlet**

### HttpServlet概述

HttpServlet类是GenericServlet的子类，它提供了对HTTP请求的特殊支持，所以通常我们都会通过继承HttpServlet来完成自定义的Servlet。

1、是一个Servlet，继承自Genericservlet.针对于HTTP协议所定制。

2、在servic()方法中直接把 ServletReuqest 和ServletResponse转为HttpServletRequest 和HttpServletResponse并调用了重载的 service(HttpServletRequest,HttpservletResponse)。

3、service(HttpServletRequest,HttpServletResponse)获取了请求方式：request.getMethod().根据请求方式有创建了doxxx（)方法（xxx为具体的请求方式，比如doGet，doPost)

4、实际开发中，直接继承HttpServlet，并根据请求方式复写doXxx（）方法接口。

5、好处：直接由针对性的覆盖doXXX（）；直接使用HttpServletRequest和HttpServletReponse，不需要强转。

6、Service可以处理get/post方式的请求，如果servlet中有Service方法，会优先调用service方法进行请求处理。

7、如果在覆写的service方法中调用父类的service方法(super.service(arg0,arg1)),则service方法处理完后，会再次根据请求方式响应的doGet和doPost方法执行。所以，一般情况下我们是不在覆写的service中调用父类的service方法，避免出现405。

### HttpServlet覆盖了service()方法

HttpServlet类中提供了service(HttpServletRequest,HttpServletResponse)方法，这个方法是HttpServlet自己的方法，不是从Servlet继承来的。在HttpServlet的service(ServletRequest,ServletResponse)方法中会把ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse，然后调用service(HttpServletRequest,HttpServletResponse)方法，这说明子类可以去覆盖service(HttpServletRequest,HttpServletResponse)方法即可，这就不用自己去强转请求和响应对象了。

其实子类也不用去覆盖service(HttpServletRequest,HttpServletResponse)方法，因为HttpServlet还要做另一步简化操作，下面会介绍。

|  |
| --- |
| **public** **abstract** **class** HttpServlet **extends** GenericServlet {    @Override  **public** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException {  HttpServletRequest request;  HttpServletResponse response;  **try** {  request = (HttpServletRequest) req;//增加了关于http协议相关的数据。  response = (HttpServletResponse) res;  } **catch** (ClassCastException e) {  **throw** **new** ServletException("non-HTTP request or response");  }  service(request, response);  }  **protected** **void** service(HttpServletRequest req, HttpServletResponse resp)  **throws** ServletException, IOException {  //这里以下部分对get或者时post请求进行判断，判断之后调用相应的方法。  ……  }  ……  }所以，我们在重写方法时，中需要重写doget或者是dopost两个方法。服务器会自动调用父类的service，并在该service中调用对应的http协议相关的service，然后这个service会调用的，子类的dopost或者是doget方法。 |

### doGet()和doPost()

在HttpServlet的service(HttpServletRequest,HttpServletResponse)方法会去判断当前请求是GET还是POST，如果是GET请求，那么会去调用本类的doGet()方法，如果是POST请求会去调用doPost()方法，这说明我们在子类中去覆盖doGet()或doPost()方法即可。

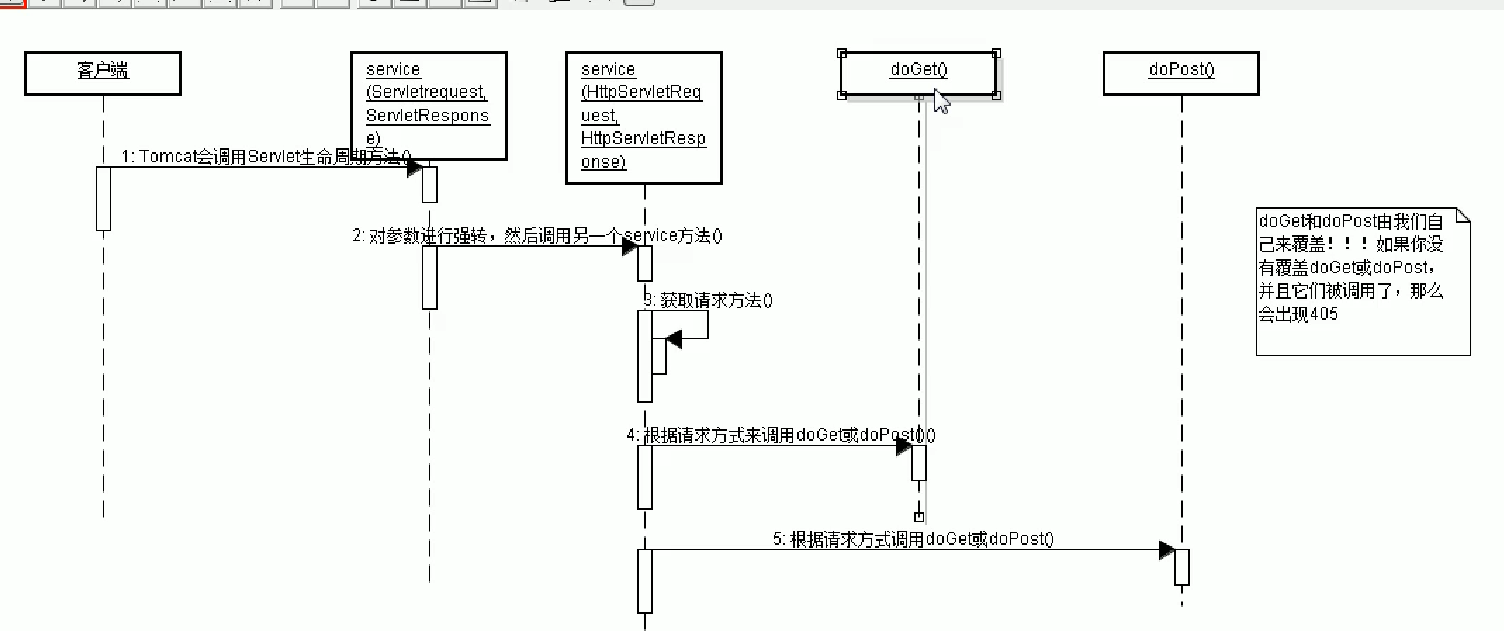
1、浏览器发送请求到HttpSevr类调用HttpServ的service（servletRequest, servletReponse）方法。

2、由于没有找到这个方法，去调用父类（HttpServlet） 的同名方法。

3、父类的service方法将ServletRequest req请求转换成HttpServletRequest请求，再去调用service(request, response) 方法。

4、 调用的service(request, response) 方法功能是判断用户发出是什么请求，如果是get则调用子类（HttpSevr）的doGet方法，如果是post则调用子类（HttpSevr）的doPost方法。所以在子类中只需要doget和dopost方法即可。

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("hello doGet()...");  }  } |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("hello doPost()...");  }  } |



注：405客户端错误：不支持该改中请求方式。http请求的方式为get，但是代码写在了post中，会报该错误

### request转为httprequest

public void service(ServletRequest req,ServletResponse res)

throws ServletException,IOException{

HttpServletRequest request；

HttpServletResponse response;

try{

request=(HttpServletRequest)req;

response=(HttpServletResponse)res；

}catch(ClassCastException e){

throw new ServletException(\"non-HTTP request or response\")；

service(request,response)；

public void service(HttpServletRequest request,HttpServletResponse response)

throws ServletException,IOException{

//1.获取请求方式。

String method=rehuest.getMethod()；

//2.根据请求方式再调用对应的处理方法

if(\"GET\".equalsIgnoreCase(method)){

doGet(request,response)；

}else if(\"poST\".equalsIgnorecase(method)){

doPost(request,response)；

}

public void doPost(HttpServletRequest request,HttpServletResponse response)

throws ServletException,IOException{

public void doGet(HttpServletRequest request,HttpServletResponse response)

throws ServletException,IOException{

}

## response对象

 response是Servlet.service方法的一个参数，类型为javax.servlet.http.HttpServletResponse。在客户端发出每个请求时，服务器都会创建一个response对象，并传入给Servlet.service()方法。response对象是用来对客户端进行响应的，这说明在service()方法中使用response对象可以完成对客户端的响应工作。

服务器处理请求的流程：

服务器每次收到请求时，都会为这个请求开辟一个新的线程。服务器会把客户端的请求数据封装到request对象中，request就是请求数据的载体！（袋子）服务器还会创建response对象，这个对象当客户端连接在一起，它可以用来向客户端发送响应。（手机）

### response对象的功能

分为以下四种：(查看response的API方法)

1.设置响应头信息；

>setHeader(String name, String value)：例如：response.setHeader("a", "A");

2.发送状态码；

> sendError(int sc) --> 发送错误状态码，例如404、500

> sendError(int sc, String msg) --> 也是发送错误状态码，还可以带一个错误信息！

> setStatus(int sc) --> 发送成功的状态码，可以用来发送302

3.设置响应正文，响应体可以是html也可以是图片。

4.重定向；

> sendRedirect(String location)方法，重定向一定要带localtion

### 设置状态码（HttpServletResponse）

方法：> sendError(int sc) --> 发送错误状态码，例如404、500

> sendError(int sc, String msg) --> 也是发送错误状态码，还可以带一个错误信息！

> setStatus(int sc) --> 发送成功的状态码，可以用来发送302

### 设置响应头信息（设置键值对）

Content-Type 、Refresh 、Location

可以使用response对象的setHeader()方法来设置响应头！使用该方法设置的响应头最终会发送给客户端浏览器！这里设置的键是Http协议中规定的键，不能随便写，浏览器会接收到响应头信息进行解析。 例如：

setHeader(String name,String value)使用与单值的响应头。在响应头中添加响应信息，但是是同键会覆盖。最常用。

例如：response.setHeader(“aa”,”aaa”);

addHeader(String name ,String value) 在响应头中添加响应信息，但是不会覆盖。

例如：response.setHeader(“aa”,”aaa”);

response.setHeader(“aa”,”aaaa”);//为一个响应头添加多个值。

setDateHeader(String name,long value):适用于单值的毫秒类型的响应头

例如:response.setDateHeader("expires",1000\*60\*60\*24):设置浏览器缓存的时间。毫秒

response.setDateHeader("expires",0):不设置缓存。

response.setContentType("text/html;charset=utf-8" 设置响应编码格式，所有的html是一致的。

注：<meta>标签可以代替响应头：<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">

### response响应正文(实体，反馈给用户界面)

response是响应对象，向客户端输出响应正文（响应体）可以使用response的响应流，repsonse一共提供了两个响应流对象：

 PrintWriter out = response.getWriter()：发送字符相关的东西

 ServletOutputStream out = response.getOutputStream()：发送字节相关的东西。

注：这两个流不能同时存在，只能使用一个。如果同时存在，会报非法状态异常。

当然，如果响应正文内容为字符，那么使用response.getWriter()，如果响应内容是字节，例如下载时，那么可以使用response.getOutputStream()。

注意，在一个请求中，不能同时使用这两个流！也就是说，要么你使用repsonse.getWriter()，要么使用response.getOutputStream()，但不能同时使用这两个流。不然会抛出IllegalStateException异常。

字符响应流：

在使用response.getWriter()时需要注意默认字符编码为ISO-8859-1，如果希望设置字符流的字符编码为utf-8，可以使用response.setCharaceterEncoding(“utf-8”)来设置。这样可以保证输出给客户端的字符都是使用UTF-8编码的！

但客户端浏览器并不知道响应数据是什么编码的！如果希望通知客户端使用UTF-8来解读响应数据，那么还是使用response.setContentType("text/html;charset=utf-8")方法比较好，因为这个方法不只会调用response.setCharaceterEncoding(“utf-8”)，还会设置content-type响应头，客户端浏览器会使用content-type头来解读响应数据。

设置响应编码格式：response.setContentType("text/html;charset=utf-8")

|  |
| --- |
| 1.设置请求编码格式  request.setCharacterEncoding("utf-8");  2.设置响应编码格式  response.setContentType("text/html;charset=utf-8")  3.获取请求数据  4.处理请求数据  5.响应处理结果  PrintWriter out = response.getWriter();  out.print();  ServletOutputStream out = response.getOutputStream();  out.print();、  //可以将字符变为字节传递。  String s = "hello world";  byte[] bytes = s.getBytes();  response.getOutputStream().write(bytes);//打印到网页上。 |

### 设置状态码及其他方法

 response.setContentType("text/html;charset=utf-8")：等同与调用response.setHeader(“content-type”, “text/html;charset=utf-8”)；

text/html：表示发送的html格式，浏览器会自动解析。

text/plain：表示发的是文本格式，不会做任何操作。

text/xml：表示发的是xml格式，使用xml解析。

注：所给的格式类型应与给的数据类型一致。

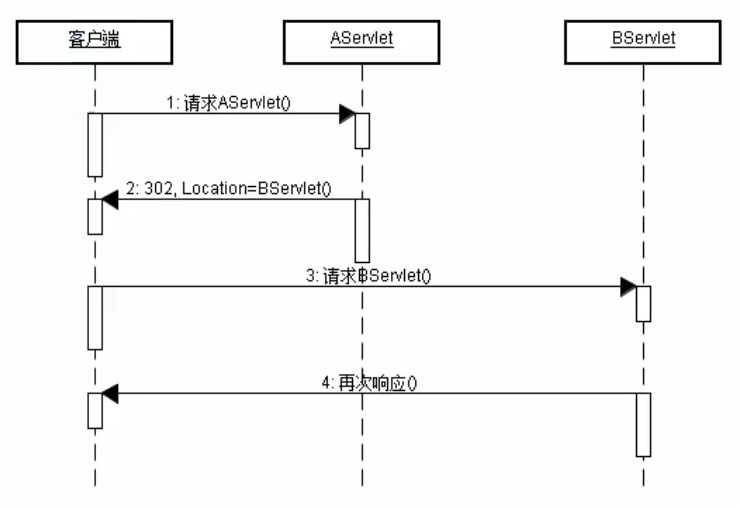
 response.setCharacterEncoding(“utf-8”)：设置字符响应流的字符编码为utf-8；

 response.setStatus(200)：设置状态码；

 response.sendError(404, “您要查找的资源不存在”)：当发送错误状态码时，Tomcat会跳转到固定的错误页面去，但可以显示错误信息。

### 重定向（）

发送302，设置localtion头，完成重定向。



sendRedirect重定向的方法也实质是设置了状态码和Location。

步骤：

1、设置Location

2、发送302状态吗

用户请求Bservlet ，然后Bservlet响应302，给出localtion头。浏览器重定向到Aservlet

response.setHeader("Location", "/servlet/test");//项目名加servlet的访问名。

response.setStatus(302);设置状态码为302，重定向。

### 定时刷新

无论是响应头、响应体、和状态码都是response发出的。

设置名为refresh的响应头

response.setHeader("refresh", "3;url=/web2/show.jsp");3秒后刷新

## Request对象

request是Servlet.service()方法的一个参数，类型为javax.servlet.http.HttpServletRequest。在客户端发出每个请求时，服务器都会创建一个request对象，并把请求数据封装到request中，然后在调用Servlet.service()方法时传递给service()方法，这说明在service()方法中可以通过request对象来获取请求数据。

数据格式：

请求行：

请求头

空行

请求体（Get没体）

### request的功能

1.封装了请求头数据；

2.封装了请求正文数据，如果是GET请求，那么就没有正文；

3.request是一个域对象，可以把它当成Map来添加获取数据；

4.request提供了请求转发和请求包含功能；

### request域方法

request是域对象！在JavaWeb中一共四个域对象，其中ServletContext就是域对象，它在整个应用中只创建一个ServletContext对象。request其中一个，request可以在一个请求中共享数据。

一个请求会创建一个request对象，如果在一个请求中经历了多个Servlet，那么多个Servlet就可以使用request来共享数据。现在我们还不知道如何在一个请求中经历之个Servlet，后面在学习请求转发和请求包含后就知道了。

request的域方法：

void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在request中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；

 Object getAttribute(String name)：用来获取request中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)request.getAttribute(“xxx”);，获取名为xxx的域属性；

 void removeAttribute(String name)：用来移除request中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；

 Enumeration getAttributeNames()：获取所有域属性的名称；

### request获取请求头数据

>String getHeader(String name),适用于单值头(最常用)

>int getIntHeader(String name),适用于单值int类型的请求头

>long getDatelHeader(String narme),适用于单值毫秒类型的请求头

>Enumeration<String>getHeaders(String name),适用于多值请求头

request与请求头相关的方法有：通过键获取值。

 String getHeader(String name)：获取指定名称的请求头；

例如：String value = req.getHeader(“User-Agent”);

如果指定的name在请求头中不存在，则会返回null。

 Enumeration e = req.getHeaderNames(“多值”);

while(e.hasMoreElements()){

String key = (String) e.nextElement();

String value = req.getHeader(key);

System.out.println(key+":"+value);

}通过迭代，获取键和值

|  |
| --- |
| //获取客户端的IP地址、获取请求方式、获取uger-Agent，得到户端的信息（操作系统浏览器）。  String addr = request.getRemoteAddr();  System.err.println("Ip:"+addr);  System.out.println("请求方式:"+request.getMethod());//获取请求方式  String userAgent = request.getHeader("User-Agent");  System.out.println(userAgent);    结果：  Ip:127.0.0.1  请求方式:GET //只有表单中的METHOD的方式为Post,才为POST请求。  Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.12 Safari/537.36  //通过这个可以获取采取的什么浏览器，进行判断。 |



### request获取请求相关的其它方法

request中还提供了与请求相关的其他方法，有些方法是为了我们更加便捷的方法请求头数据而设计，有些是与请求URL相关的方法。

 int getContentLength()：获取请求体的字节数，GET请求没有请求体，没有请求体返回-1；

 String getContentType()：获取请求类型，如果请求是GET，那么这个方法返回null；如果是POST请求，那么默认为application/x-www-form-urlencoded，表示请求体内容使用了URL编码；

 String getMethod()：返回请求方法，例如：GET

 Locale getLocale()：返回当前客户端浏览器的Locale。java.util.Locale表示国家和言语，这个东西在国际化中很有用；

 String getCharacterEncoding()：获取请求编码，如果没有setCharacterEncoding()，那么返回null，表示使用ISO-8859-1编码；

 void setCharacterEncoding(String code)：设置请求编码，只对请求体有效！注意，对于GET而言，没有请求体！！！所以此方法只能对POST请求中的参数有效！

 String getContextPath()：返回上下文路径，例如：/hello

 String getQueryString()：返回请求URL中的参数，例如：name=zhangSan

 String getRequestURI()：返回请求URI路径，例如：/hello/oneServlet

 StringBuffer getRequestURL()：返回请求URL路径，例如：http://localhost/hello/oneServlet，即返回除了参数以外的路径信息；

 String getServletPath()：返回Servlet路径，例如：/oneServlet

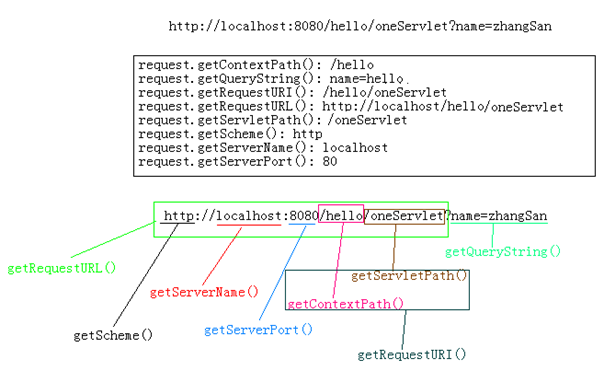
 String getRemoteAddr()：返回当前客户端的IP地址；

 String getRemoteHost()：返回当前客户端的主机名，但这个方法的实现还是获取IP地址；

 String getScheme()：返回请求协议，例如：http；

 String getServerName()：返回主机名，例如：localhost

 int getServerPort()：返回服务器端口号，例如：8080



|  |
| --- |
| http://localhost:8080/day10-2/AServlet?username=xxx&password=yy  >String getScheme():获取协议，http  >String getServerName():获取服务器名，1ocalhost  >String getServerPort():获取服务器端口，8080  >String getContextPath():获取项目名，/day102  >String getServletPath():获取Servlet路径，/Aservlet  >String getQuerystring():获取参数部分，即问号后面的部分。username=xxx&password=yyy  >String getRequestURI():获取请求URI，等于项目名+Servlet路径。/day10 2/AServlet  >String getRequestURL():获取请求URL，等于不包含参数的整个请求路径。  <http://localhost:8080/day10-2/AServlet>  获取整个连接getRequestURI()+”?”+ getQuerystring() |

### 封ip

可以使用request.getRemoteAddr()方法获取客户端的IP地址，然后判断IP是否为禁用IP。

|  |
| --- |
| String ip = request.getRemoteAddr();  System.*out*.println(ip);  **if**(ip.equals("127.0.0.1")) {  response. getWriter().print("您的IP已被禁止！");  } **else** {  response.getWriter().print("Hello!");  } |

### request获取请求参数(请求数据)

1. 获取请求参数：请求参数是由客户端发送给服务器的！有可能是在请求体中（POST），也可能是在URL之后（GET）。最为常见的客户端传递参数方式有两种：（无论是哪种方式，获取的方法都相同）

请求参数：有一个参数一个值的，一个参数多个值的。都是字符串。

 浏览器地址栏直接输入：一定是GET请求；

 超链接：一定是GET请求；

 表单：可以是GET，也可以是POST，这取决与<form>的method属性值；

2 GET请求和POST请求的区别：

 GET请求：

 请求参数会在浏览器的地址栏中显示，所以不安全；

 请求参数长度限制长度在1K之内；

 GET请求没有请求体，无法通过request.setCharacterEncoding()来设置参数的编码；

 POST请求：

 请求参数不会显示浏览器的地址栏，相对安全；

 请求参数长度没有限制；

3获取方法

String getParameter(String name):获取指定名称的请求参数值，适用于单值请求参数

String[] getParaneterValuea(String name):获取指定名称的请求参数值，适用于多值请求参数

Enumeration<String>getParameterNames():获取所有请求参数名称

Map<String,String[ ]>getParameterMap():获取所有请求参数，其中Key为参数名，value为参数值。值可能为多值。

秦例：超链接参数

案例：表单数据

4例如

|  |
| --- |
| <a href=*"/hello/ParamServlet?p1=v1&p2=v2"*>超链接</a>  <hr/>  <form action=*"/hello/ParamServlet"* method=*"post"*>  参数1：<input type=*"text"* name=*"p1"*/><br/>  参数2：<input type=*"text"* name=*"p2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |
|  |

获取参数方法：

String getParameter(String name)：通过指定名称获取参数值；

|  |
| --- |
| **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String v1 = request.getParameter("p1");  String v2 = request.getParameter("p2");  System.*out*.println("p1=" + v1);  System.*out*.println("p2=" + v2);  }    **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String v1 = request.getParameter("p1");  String v2 = request.getParameter("p2");  System.*out*.println("p1=" + v1);  System.*out*.println("p2=" + v2);  } |

String[] getParameterValues(String name)：当多个参数名称相同时，可以使用方法来获取；

|  |
| --- |
| <a href=*"/hello/ParamServlet?name=zhangSan&name=liSi"*>超链接</a> |
| **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String[] names = request.getParameterValues("name");  System.*out*.println(Arrays.*toString*(names));  } |

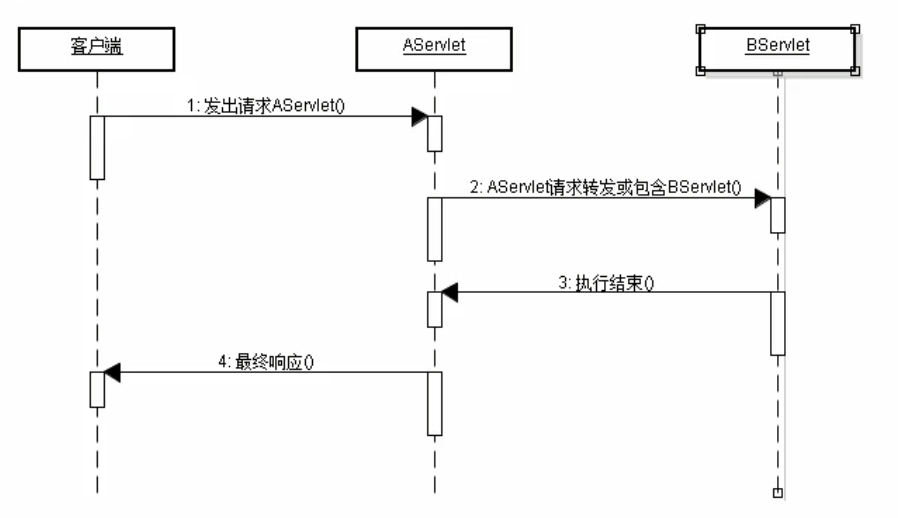
Enumeration getParameterNames()：获取所有参数的名字；

|  |
| --- |
| <form action=*"/hello/ParamServlet"* method=*"post"*>  参数1：<input type=*"text"* name=*"p1"*/><br/>  参数2：<input type=*"text"* name=*"p2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  Enumeration names = request.getParameterNames();  **while**(names.hasMoreElements()) {  System.*out*.println(names.nextElement());  }  } |

Map getParameterMap()：获取所有参数封装到Map中，其中key为参数名，value为参数值，因为一个参数名称可能有多个值，所以参数值是String[]，而不是String。

|  |
| --- |
| <a href=*"/day05\_1/ParamServlet?p1=v1&p1=vv1&p2=v2&p2=vv2"*>超链接</a> |
| Map<String,String[]> paramMap = request.getParameterMap();  **for**(String name : paramMap.keySet()) {  String[] values = paramMap.get(name);  System.*out*.println(name + ": " + Arrays.*toString*(values));  } |
| p2: [v2, vv2]  p1: [v1, vv1] |

### 请求转发



无论是请求转发还是请求包含，都表示由多个Servlet共同来处理一个请求。例如Servlet1来处理请求，然后Servlet1又转发给Servlet2来继续处理这个请求。而且还能够在Aservlet中存储一些值，在BServlet中使用，此时的request作为域对象存在

作用：实现多个servlet联动操作请求，避免代码冗余，让servlet的职责更加明确。

格式：RequestDispatcher rd= Resrequest.getRequestDispatcher("/ServletName");

含义：使用request获取.RequestDispatcher对象，方法的参数为被转发或被包含的servlet路径。

请求转发：rd.forward(request, response);（choingong）

请求包含：rd. include(request, response);

特点：一次请求，浏览器信息栏不改变。

有时一个请求需要多个servlet协作才能完成，所以需要在一个servlet跳到另一个servlet！

>一个请求跨多个servlet，需要使用转发和包含。I

>请求转发：由下一个Servlet完成响应体！当前Servlet可以设置响应头！（留头不留体）前一个servlet可以设置响应头，下个servlet可以设置响应体。即使上一个serlvet输出了内容，也不能在网页中输出。使用两个流来输出的内容为响应体。

>请求包舍：由两个servlet共同未完成响应体！（都留）都可以设置响应头和响应体。

>无论是请求转发还是请求包含，都在一个请求范围内！使用同一个request和response！

request域

在AServlet中，把请求转发到BServlet：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("AServlet");  RequestDispatcher rd = request.getRequestDispatcher("/BServlet");  rd.forward(request, response);  //System.out.println(“是否能执行”)//请求转发后的代码不会执行，其他的servket已经给出响应。一般直接return；  }  } |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("BServlet");  }  } |
| Aservlet  BServlet |

### 请求包含

在AServlet中，把请求包含到BServlet：

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("AServlet");  RequestDispatcher rd = request.getRequestDispatcher("/BServlet");  rd.include(request, response);  }  } |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("BServlet");  }  } |
| Aservlet  BServlet |

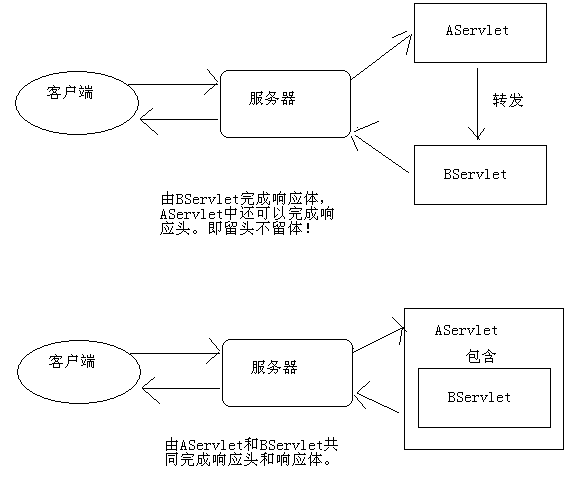
### 请求包含和请求转发的比较

1.如果在AServlet中请求转发到BServlet，那么在AServlet中就不允许再输出响应体，即不能再使用response.getWriter()和response.getOutputStream()向客户端输出，这一工作应该由BServlet来完成；如果是使用请求包含，那么没有这个限制；(留头不留体)

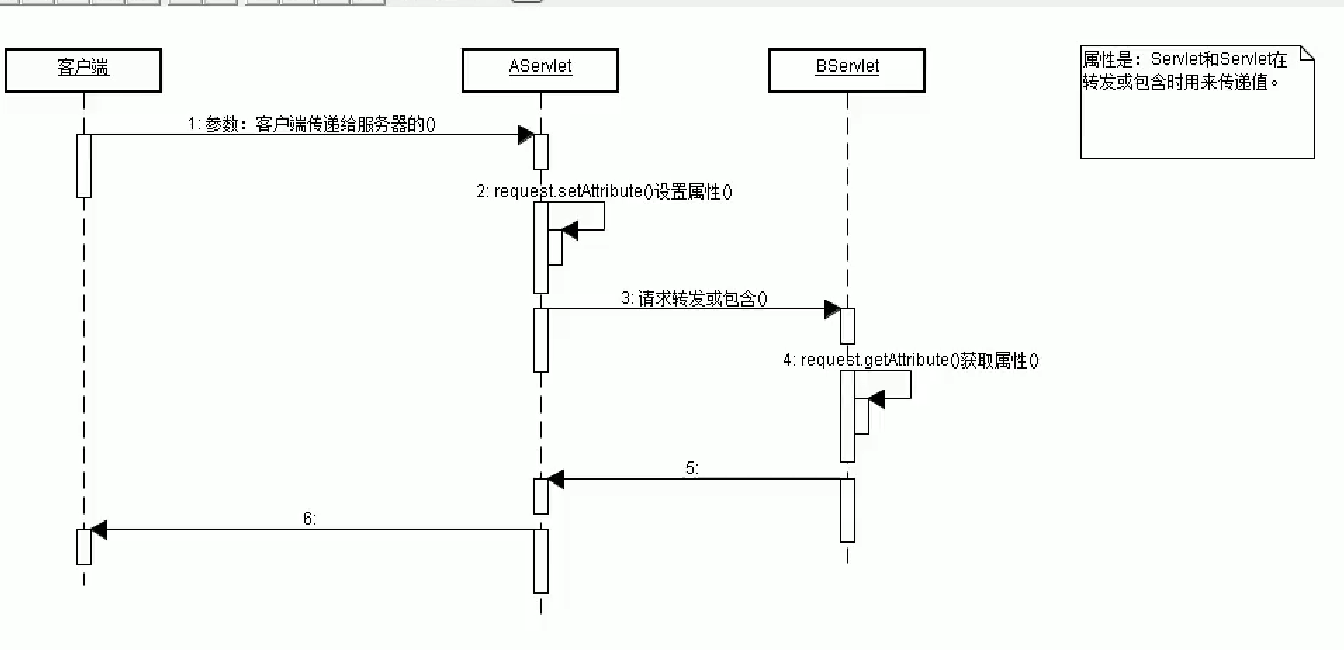
 2.请求转发虽然不能输出响应体，但还是可以设置响应头的，例如：response.setContentType(”text/html;charset=utf-8”);

 3.请求包含大多是应用在JSP页面中，完成多页面的合并；(留头也留体)

 4.请求请求大多是应用在Servlet中，转发目标大多是JSP页面；



### request域对象



引入：使用请求转发后，不同Servlet之间怎么进行数据的共享？或者说数据怎么从一个servlet流转给另外一个Servlet呢？

解决方法：使用request作用域。

作用域：一次请求内。在AServlet设置一个属性，转发或包含到BServlet中，在BServlet中可以获取到该属性值。

作用：解决了一次请求内不同Servlet的数据(请求数据+其他数据)共享问题。

request是域对象！在JavaWeb中一共四个域对象，其中ServletContext就是域对象，它在整个应用中只创建一个ServletContext对象。request其中一个，request可以在一个请求中共享数据。

一个请求会创建一个request对象，如果在一个请求中经历了多个Servlet，那么多个Servlet就可以使用request来共享数据。现在我们还不知道如何在一个请求中经历之个Servlet，后面在学习请求转发和请求包含后就知道了。

下面是request的域方法：

void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在request中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；

 Object getAttribute(String name)：用来获取request中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)request.getAttribute(“xxx”);，获取名为xxx的域属性；

 void removeAttribute(String name)：用来移除request中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；

 Enumeration getAttributeNames()：获取所有域属性的名称；

### 请求转发和重定向的区别

>请求转发是一个请求一次响应，而重定向是两次请求两次响应

>请求转发地址栏不变化，而重定向会显示后一个请求的地址

>请求转发只能转发到本项目其他Servlet，而重定向不只能重定向到本项目的其他servlet，还能定向到其他项目

>请求转发是服务器端行为，只需给出转发的servlet路径，而重定向需要给出requestURI

，即包含项目名！

>请求转发和重定向效率是转发高！因为是一个请求！

<>需要地址栏发生变化，那么必须使用重定问！

<>需要在下一个servlet中获取requeat域中的数据，必须要使用转发！

## servlet细节

### 让服务器在启动时就创建Servlet

默认情况下，服务器会在某个Servlet第一次收到请求时创建它。也可以在web.xml中对Servlet进行配置，使服务器启动时就创建Servlet。

|  |
| --- |
| <servlet>  <servlet-name>hello1</servlet-name>  <servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>  <load-on-startup>0</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>hello1</servlet-name>  <url-pattern>/hello1</url-pattern>  </servlet-mapping>  <servlet>  <servlet-name>hello2</servlet-name>  <servlet-class>cn.itcast.servlet.Hello2Servlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>hello2</servlet-name>  <url-pattern>/hello2</url-pattern>  </servlet-mapping>  <servlet>  <servlet-name>hello3</servlet-name>  <servlet-class>cn.itcast.servlet.Hello3Servlet</servlet-class>  <load-on-startup>2</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>hello3</servlet-name>  <url-pattern>/hello3</url-pattern>  </servlet-mapping> |

在<servlet>元素中配置<load-on-startup>元素可以让服务器在启动时就创建该Servlet，其中<load-on-startup>元素的值必须是大于等于的整数，它的使用是服务器启动时创建Servlet的顺序。上例中，根据<load-on-startup>的值可以得知服务器创建Servlet的顺序为Hello1Servlet、Hello2Servlet、Hello3Servlet。

### Servlet与线程安全

因为一个类型的Servlet只有一个实例对象，那么就有可能会现时出一个Servlet同时处理多个请求，那么Servlet是否为线程安全的呢？答案是：“不是线程安全的”。这说明Servlet的工作效率很高，但也存在线程安全问题！

所以我们不应该在Servlet中随便创建成员变量，因为可能会存在一个线程对这个成员变量进行写操作，另一个线程对这个成员变量进行读操作。

### <url-pattern>

<url-pattern>是<servlet-mapping>的子元素，用来指定Servlet的访问路径，即URL。它必须是以“/”开头！

1.可以在<servlet-mapping>中给出多个<url-pattern>，例如：

|  |
| --- |
| <servlet-mapping>  <servlet-name>AServlet</servlet-name>  <url-pattern>/AServlet</url-pattern>  <url-pattern>/BServlet</url-pattern>  </servlet-mapping> |

那么这说明一个Servlet绑定了两个URL，无论访问/AServlet还是/BServlet，访问的都是AServlet。

2、还可以在<url-pattern>中使用通配符，所谓通配符就是星号“\*”，星号可以匹配任何URL前缀或后缀，使用通配符可以命名一个Servlet绑定一组URL，例如：

 <url-pattern>/servlet/\*<url-patter>：/servlet/a、/servlet/b，都匹配/servlet/\*；

 <url-pattern>\*.do</url-pattern>：/abc/def/ghi.do、/a.do，都匹配\*.do；

 <url-pattern>/\*<url-pattern>：匹配所有URL；

请注意，通配符要么为前缀，要么为后缀，不能出现在URL中间位置，也不能只有通配符。例如：/\*.do就是错误的，因为星号出现在URL的中间位置上了。\*.\*也是不对的，因为一个URL中最多只能出现一个通配符。

注意，通配符是一种模糊匹配URL的方式，如果存在更具体的<url-pattern>，那么访问路径会去匹配具体的<url-pattern>。例如：

|  |
| --- |
| <servlet>  <servlet-name>hello1</servlet-name>  <servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>hello1</servlet-name>  <url-pattern>/servlet/hello1</url-pattern>  </servlet-mapping>  <servlet>  <servlet-name>hello2</servlet-name>  <servlet-class>cn.itcast.servlet.Hello2Servlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>hello2</servlet-name>  <url-pattern>/servlet/\*</url-pattern>  </servlet-mapping> |

　当访问路径为http://localhost:8080/hello/servlet/hello1时，因为访问路径即匹配hello1的<url-pattern>，又匹配hello2的<url-pattern>，但因为hello1的<url-pattern>中没有通配符，所以优先匹配，即设置hello1。

### web.xml文件的继承（了解）

在${CATALINA\_HOME}\conf\web.xml中的内容，相当于写到了每个项目的web.xml中，它是所有web.xml的父文件。

每个完整的JavaWeb应用中都需要有web.xml，但我们不知道所有的web.xml文件都有一个共同的父文件，它在Tomcat的conf/web.xml路径。

conf/web.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"ISO-8859-1"*?>  <web-app xmlns=*"http://java.sun.com/xml/ns/javaee"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee*  *http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"*  version=*"3.0"*>  <servlet>  <servlet-name>default</servlet-name>  <servlet-name>/</servlet-name>  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>  <init-param>  <param-name>debug</param-name>  <param-value>0</param-value>  </init-param>  <init-param>  <param-name>listings</param-name>  <param-value>false</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet>  <servlet-name>jsp</servlet-name>  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>  <init-param>  <param-name>fork</param-name>  <param-value>false</param-value>  </init-param>  <init-param>  <param-name>xpoweredBy</param-name>  <param-value>false</param-value>  </init-param>  <load-on-startup>3</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>jsp</servlet-name>  <url-pattern>\*.jsp</url-pattern>  <url-pattern>\*.jspx</url-pattern>  </servlet-mapping>  <session-config>  <session-timeout>30</session-timeout>  </session-config>  <!-- 这里省略了大概4000多行的MIME类型的定义,这里只给出两种MIME类型的定义 -->  <mime-mapping>  <extension>bmp</extension>  <mime-type>image/bmp</mime-type>  </mime-mapping>  <mime-mapping>  <extension>htm</extension>  <mime-type>text/html</mime-type>  </mime-mapping>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>index.jsp</welcome-file>  </welcome-file-list>  </web-app> |
|  |

### servlet常见错误

1、404错误：资源未找到

1.请求地址中的servlet的别名书写错误。

2.虚拟项目名称书写错误。

2、500错误：内部服务器错误

1.ClassbNotFoundException

1.在web.xml中校验servlet类的全限定路径是否拼写错误。

2.因为service方法的代码执行错误导致。

3、405错误：请求方式不支持

1. 请求方式和servlet中的方法不匹配，尽量使用service方法进行请求处理，并不要在service方法中调用父类的service。

## 乱码问题

常见字符编码：iso-8859-1(不支持中文)、gb2312、gbk、gb18030(系统默认编码，中国的国标码)、utf-8(万国码，支持全世界的编码，所以我们使用这个)

### 响应编码

当使用response.getWriter()来向客户端发送字符数据时，如果在之前没有设置编码，那么默认使用iso，因为iso不支持中文，一定乱码。

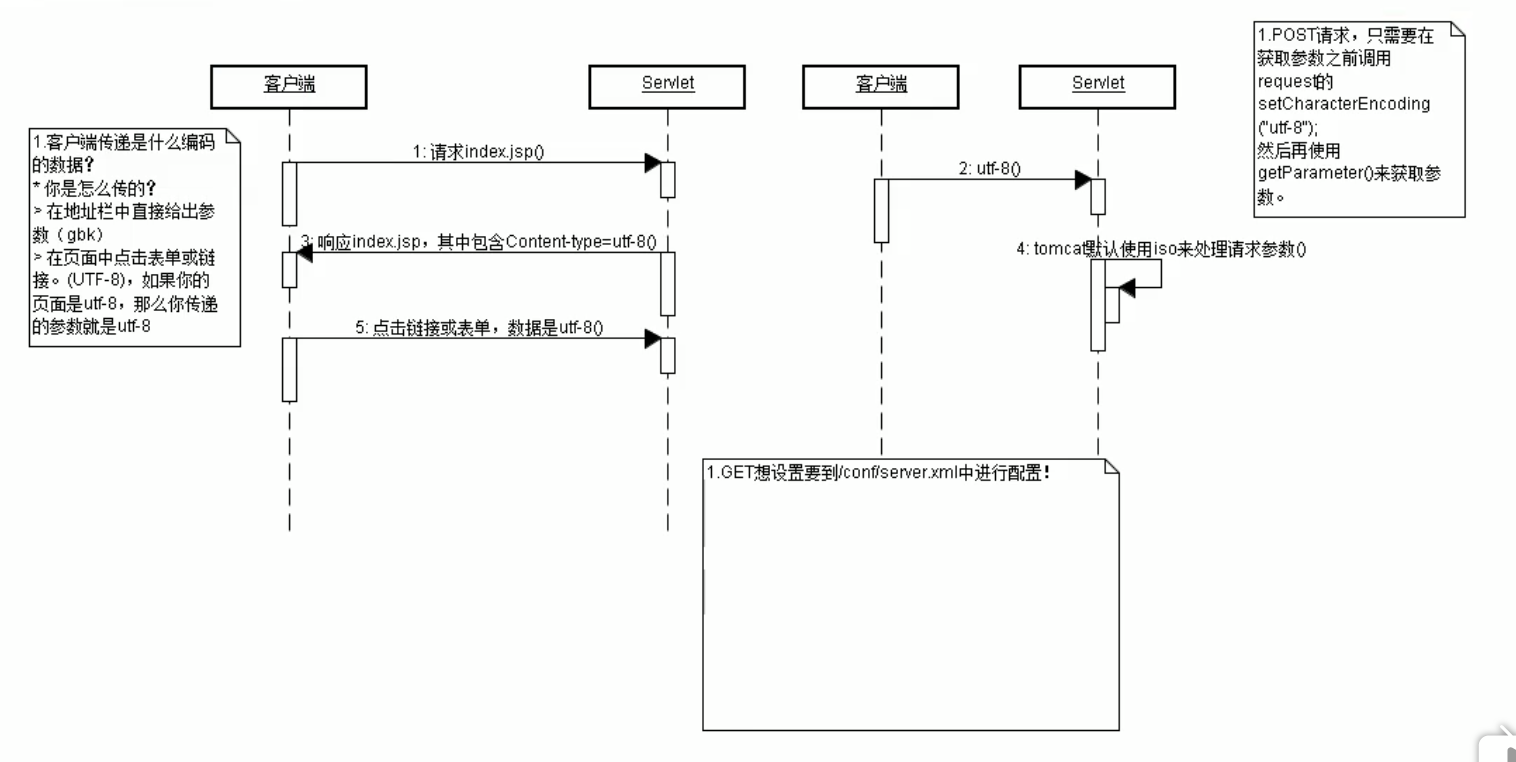
解决方法：在使用response.getWriter()之前可以使用response.setCharaceterEncoding()来设置字符流的编码为gbk或utf-8，当然我们通常会选择utf-8。这样使用response.getWriter()发送的

字符就是使用utf-8编码的。但还是会出现乱码！因为浏览器并不知道服务器发送过来的是什么编码的数据！这时浏览器会使用gbk来解码，所以乱码！

使用response.getWriter()之前可以使用response.setHeader("Content-type","text/html;charset=utf-8")来设置响应头，通知浏览器服务器这边使用的是utf-8编码，而且在调用setHeader()后，还会自动执行setCharacterEncding()方法。这样浏览器会使用utf-8解码，所以就不会乱码了！

setHeader("Content-Type", "text/html;charset=utf-8")的快捷方法是：setContentType("text/html;charset=utf-8)。

### 请求编码



客户端发送给服务器的请求参数是什么编码：

客户端首先要打开一个页面，然后在页面中提交表单或点击超链接！在请求这个页面时，服务器响应的编码是什么，那么客户端发送请求时的编码就是什么。

服务器端默认使用什么编码来解码参数：

　 服务器端默认使用ISO-8859-1来解码！所以这一定会出现乱码的！因为iso不支持中文！

　　请求编码处理分为两种：GET和POST：GET请求参数不在请求体中，而POST请求参数在请求体中，所以它们的处理方式是不同的！

　　\* GET请求编码处理：

　　　　> String username = new String(request.getParameter("iso-8859-1"), "utf-8");

　　　 > 在tomcat/config/server.xml中配置Connector标签添加URIEncoding="UTF-8"

　　\* POST请求编码处理：

　　　　> String username = new String(request.getParameter("iso-8859-1"), "utf-8");

　　　　> 在获取参数之前调用request.setCharacterEncoding("utf-8");

//放在代码的第一行。

注：通过上述方法都可以解决中文乱码问题，但是在代码维护方面却有很大麻烦，因

为它不仅增加了代码量，而且移植性也差。

可以通过编写一个Servlet过滤器来解决中文乱码问题，并可以通过配置让过滤器解决所有请求处理字符集的问题，请求处理页面就不用关心字符集的处理了。这种方法可以有效防止上述问题且移植性强。

### URL编码

　　表单的默认类型：Content-Type: application/x-www-form-urlencoded，就是把中文转换成%后面跟随两位的16进制。

　　为什么要用它：在客户端和服务器之间传递中文时需要把它转换成网络适合的方式。

　　\* 它不是字符编码！

　　\* 它是用来在客户端与服务器之间传递参数用的一种方式！

　　\* URL编码需要先指定一种字符编码，把字符串解码后，得到byte[]，然后把小于0的字节+256，再转换成16进制。前面再添加一个%。

　　\* POST请求默认就使用URL编码！tomcat会自动使用URL解码！

　　\* URL编码：String username = URLEncoder.encode(username, "utf-8");编码（所有超链接的中文都要使用URL来编码）

　　\* URL解码：String username = URLDecoder.decode(username, "utf-8");

　　最后我们需要把链接中的中文参数，使用url来编码！今天不行，因为html中不能给出java代码，但后面学了jsp就可以了。

### 路径

1. web.xml中<url-pattern>路径，（叫它Servlet路径！）要么以“\*”开关，要么为“/”开

　　　　>头

2.转发和包含路径(服务器端路径)

　 以“/”开头：相对当前项目路径，（一般使用这个，当修改路径之后，这个路径不会变）

例如：http://localhost:8080/项目名/　request.getRequestdispacher("/BServlet").for...();

　 不以“/”开头：相对当前Servlet路径。

request.getRequestdispacher("/BServlet").for...();，假如当前Servlet是：http://localhost:8080/项目名/servlet/AServlet，

就是http://localhost:8080/项目名/servlet/BServlet

　3.重定向路径（客户端路径）

　 以“/”开头：相对当前主机，例如：http://localhost:8080/，　所以需要自己手动添加项目名，例如；response.sendRedirect("/day10\_1/Bservlet");没写项目名会报404.

　4.页面中超链接和表单路径

　　　　> 与重定向相同，都是客户端路径！需要添加项目名

> <form action="/day10\_1/AServlet">

> <a href="/day10\_/AServlet">

> <a href="AServlet">，如果不以“/”开头，那么相对当前页面所在路径。如果是http://localhost:8080/day10\_1/html/form.html。

http://localhost:8080/day10\_1/html/ASevlet

> \*\*\*\*\*建立使用以“/”开头的路径，即绝对路径！

　5. ServletContext获取资源路径

　　　　> 相对当前项目目录，即当然index.jsp所在目录

　6. ClassLoader获取资源路径，加载类路径，相当于src的路径

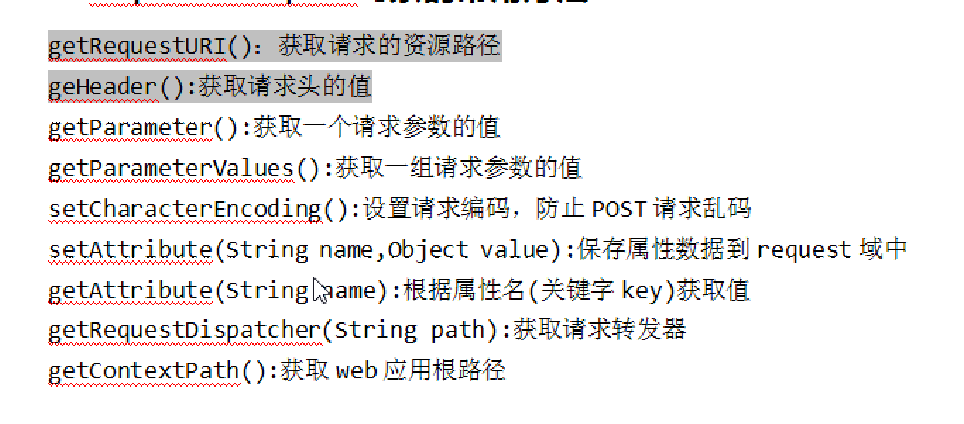
　　　　> 相对classes目录

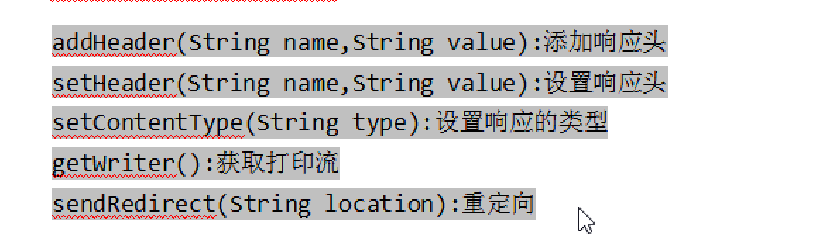
　　\* Class获取资源路径

　　　　> 以“/”开头相对classes目录

　　　　> 不以“/”开头相对当前.class文件所在目录。

## 常用方法



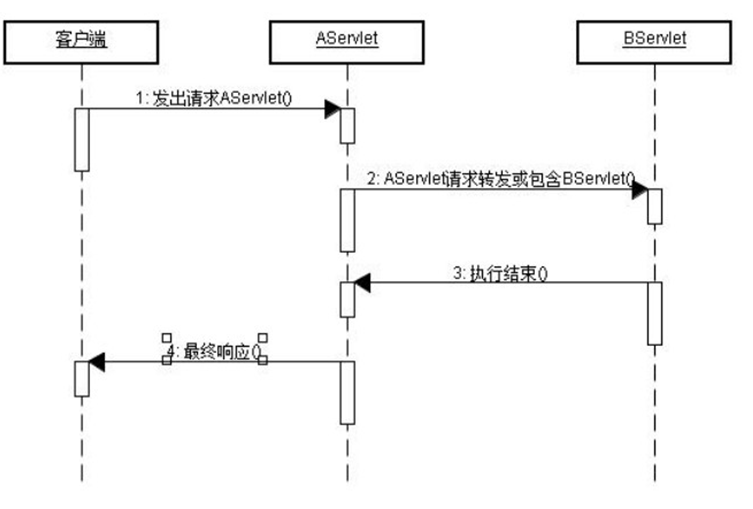


## 请求重定向和转发

### 请求转发

多个Servlet 共同来处理一个请求

请求转发过程：浏览器向服务器发送一个请求（发送到web容器），服务器第一次访问AServlet时创建AServlet，并创建ServletRequest和ServletRsponse，然后传递给Aservlet，Aservlet读取请求信息和写入响应信息，发出forward指令给了web容器，然后服务器继续给访问BServlet，重复访问操作，最后服务器将响应传递给客户端。



(1)调用HttpServletRequest的getRequestDispatcher()方法获取RequestDispatcher 对象调用 getRequestDispatcher（)需要传入要转发的地址。

String path = “testServlet”;

RequestDispatcher requestDispatcher=request.getRequestDispatcher(“/”+path);

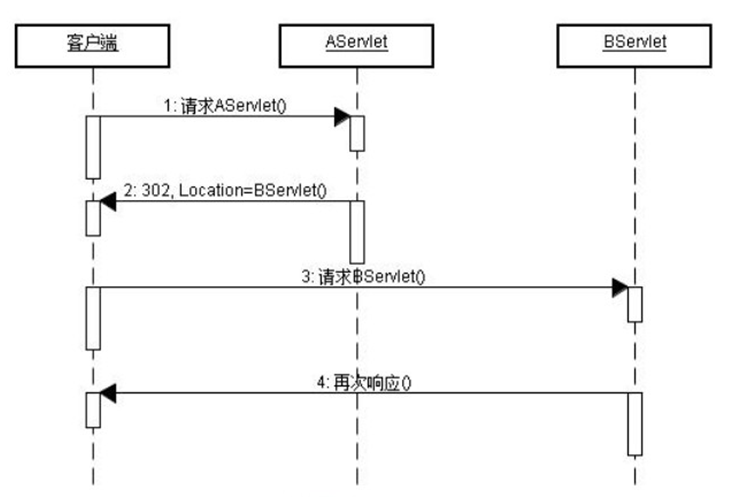
(2)调用HttpServletRequest的forward(request,response)进行请求的转发。

requestDispatcher.forward(request,response);|

(3) request.getRequestDispatcher(path).froward.(request.response);

### 请求重定向

浏览器向服务器发送一个请求（发送到web容器），服务器第一次访问AServlet时创建AServlet，并创建ServletRequest和ServletRsponse，然后传递给Aservlet，Aservlet读取请求信息和写入响应信息返回服务器，服务器将Http响应传递给客户端。客户端继续发出请求响应Bservlet。



执行请求的重定向，直接调用response.sendRedirect(path)方法，

path 为要重定向的地址

string path="testservlet"；

response.sendRedirect(location)；

### 请求转发和重定向的区别

1、请求的转发只发出了一次请求，重定向发出了两次请求

2、请求的转发：地址栏是初次发出请求的地址。

请求的冲定向:地址栏不再是初次发出的请求地址，地址栏为最后响应的地址。

3、请求转发：在最终的Servlet中，request 对象和中转的那个request是同一个对象。

请求的重定向：在最终的 Servlet中，request 对象和中转的那个request不是同一个对象。

4、请求的转发：只能转发给当前WEB应用的的资源

请求的重定向：可以重定向到任何资源。

5、请求的转发：/代表的是当前WEB应用的根目录

请求的重定向：/代表的是当前WEB站点的根目录。

### 路径问题

①绝对路径与相对路径的区分：

绝对路径为前面加上了“/”的路径，而相对路径则前面没有加上“/”。

②转发与重定向中的具体讨论：

        转发：转发当中使用带“/”的绝对路径时，表示的是当前web应用的根目录再加上传入的目录，因为转发是在服务器内部进行的,写绝对路径/开头指的是当前的Web应用程序

   eg: 假设当前的web应用目录为http://localhost:8080/web-app  执行req.getRequestDispatcher("/login.jsp")  则这个时候转到的目录为 http://localhost:8080/web-app/login.jsp

  而当转发中使用的是不带“/”的相对路径的时候，表示的是当前的路径再加上传入的路径

  eg:假如当前的路径为http://localhost:8080/web-app/abc  执行req.getRequestDispatcher("login.jsp") 则这个时候转到的目录为  http://localhost:8080/web-app/abc/login.jsp

        重定向：重定向中使用带“/”的绝对路径时，表示的是当前服务器的根目录再加上传入的目录！

 eg： 假设当前的web应用目录为http://localhost:8080/web-app 执行   res.sendRedirect("/login.jsp") 则这个时候转到的目录为   http://localhost:8080/login.jsp

       而当转发中使用的是不带“/”的相对路径的时候，表示的是当前的路径再加上传入的路径(和转发的时候一样)

 eg:假如当前的路径为http://localhost:8080/web-app/abc 执行req.getRequestDispatcher("login.jsp") 则这个时候转到的目录为  <http://localhost:8080/web-app/abc/login.jsp、>

## 访问路径问题

### Servlet访问路径

项目名+servlet名为请求ui

Servlet的访问路径是在web.xml文件中配置的，如下所示：

<url-pattern>Servlet访问路径</url-pattern>

Servlet访问路径有如下两种书写方式：

路径以“/”开头

路径以“\*“开头

Servlet的访问路径要么以“/”开头，要么以“\*”开头，但是在Servlet中，现在很少使用“\*”开头，在后面将要学习的过滤器中会使用它来匹配要过滤的资源。

### 请求转发和请求包含路径

请求转发和请求包含需要给出即将转发和包含资源的路径，例如：

request.getRequestDispatcher(“/BServlet”).forward(request,response)，表示要转发到BServlet。request对象的getRequestDispatcher(Stringpath)方法的参数有如下两种书写方式：

以“/”开头：

相对当前项目路径，即：http://localhost:8080/项目名，例如：request.getRequestDispatcher(“/BServlet”)表示转发或包含路径为http://localhost:8080/项目名/BServlet的Servlet。

不以“/”开头：

相对当前Servlet的路径，例如：当前Servlet是AServlet，访问它的路径为http://localhost:8080/项目名/servlet/AServlet，那么request.getRequestDispatcher(“BServlet”)就相当于，转发或包含路径为http://localhost:8080/项目名/servlet/BServlet的Servlet。

以上两种书写方式中，以“/”开头的路径是绝对路径，不以“/”开头的路径是相对路径。为了以后的开发简便，建议大家采用第一种方式，即以“/”开头的路径。

### 重定向路径

进行重定向时，需要给出将要重定向的资源的路径，例如：

response.sendRedirect(“/Example/BServlet”)，表示重定向到Example项目中的访问路径为“/BServlet”的Servlet。

重定向中的路径是客户端路径，只要是客户端路径都需要加上项目名，否则找不到将要访问的资源。重定向路径有以下要求：

路径要以“/”开头

以“/”开头，相对当前主机，即http://localhost:8080，所以路径上要加上项目名，如：response.sendRedirect(“/Example/Servlet”)，对应的路径就是：http://localhost:8080/Example/Servlet。

### 页面中超链接和表单路径

页面中的超链接和表单路径和重定向路径一样，也是客户端路径，因此路径中要加上项目名。例如：

<formaction=”/Example/AServlet”></form>

<ahref=”/Example/AServlet”></a>

如果超链接或者表单路径中没有加上项目名，如：<ahref=”AServlet”>，并且没有以“/”开头，那么就是相对路径，相对于当前页面所在路径。例如，超链接所在页面是index.jsp，那么index.jsp页面的路径就是：http://localhost:8080/项目名，超链接路径就是：http://localhost:8080/项目名/AServlet。如果是这样就不存在问题，但是如果页面并不是在应用的根目录下，而是根目录下的一个目录中，例如html目录中，那么index.jsp的访问路径就应该是：http://localhost:8080/项目名/html，那么再使用<ahref=”AServlet”>中的路径就会出现问题，所以建议大家使用绝对路径。

## ServletContext

### ServletContext（重要）

一个项目只有一个ServletContext对象！我们可以在N多个Servlet中来获取这个唯一的对象，使用它可以给多个Servlet传递数据！与天地同寿！！！这个对象在Tomcat启动时就创建，在Tomcat关闭时才会死去！

ServletContext获取资源路径使用的方式是getRealPath(Stringpath)，该方法获取的是应用发布到Tomcat安装目录下的webapps目录中的根目录，如：servletContext.getRealPath(“/”)返回的路径就是D:\apache-tomcat-7.0.42\webapps\项目名。

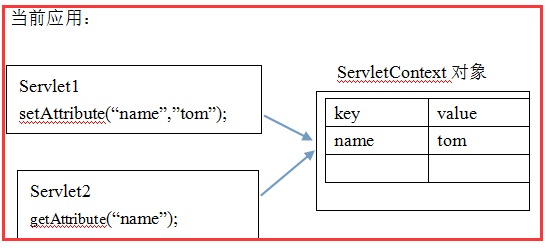
### ServletContext概述

ServletContext官方叫servlet上下文。服务器会为每一个工程创建一个对象，这个对象就是ServletContext对象。这个对象全局唯一，而且工程内部的所有servlet都共享这个对象。所以叫全局应用程序共享对象。

服务器会为每个应用创建一个ServletContext对象：

ServletContext对象的创建是在服务器启动时完成的；

ServletContext对象的销毁是在服务器关闭时完成的。



### ServletContext的作用

ServletContext对象的作用是在整个Web应用的动态资源之间共享数据！例如在AServlet中向ServletContext对象中保存一个值，然后在BServlet中就可以获取这个值，这就是共享数据了。

### 获取ServletContext

ServletConfig---------------------getServletContext()；

GenericServlet--------------------getServletContext();

//Serlvet继承了ServletConfig这个接口，所有继承实现Serlvet的类都能使用这个方法。

HttpSession------------------------getServletContext()

ServletContextEvent----------------getServletContext()

### 在Servlet中获取ServletContext对象：

在void init(ServletConfig config)中：

|  |
| --- |
| ServletContext context = config.getServletContext();，  //ServletConfig类的getServletContext()方法可以用来获取ServletContext对象；  ServletContext context=this.getServletConfig().getServletContext() |

在GenericeServlet或HttpServlet中获取ServletContext对象：

GenericServle类有getServletContext()方法，所以可以直接使用this.getServletContext()来获取；

|  |
| --- |
| public class MyServlet implements Servlet {  public void init(ServletConfig config) {  ServletContext context = config.getServletContext();  }  …  } |
| public class MyServlet extends HttpServlet {  public void doGet(HttpServletRequest request, HttpServletResponse response) {  ServletContext context = this.getServletContext();  }  } |

### 功能1：域对象

ServletConfig对象中维护了ServletContext对象的引用，开发人员在编写servlet时，可以通过ServletConfig.getServletContext方法获得ServletContext对象。由于一个WEB应用中的所有Servlet共享同一个ServletContext对象，因此Servlet对象之间可以通过ServletContext对象来实现通讯。ServletContext对象通常也被称之为context域对象。

ServletContext是JavaWeb四大域对象之一：

PageContext；

ServletRequest；

HttpSession；

ServletContext；

所有域对象都有存取数据的功能，因为域对象内部有一个Map，用来存储数据，下面是ServletContext对象用来操作数据的方法：

void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在ServletContext中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；在web项目范围内存放内容，以便让在web项目中所有的servlet读能访问到

Object getAttribute(String name)：用来获取ServletContext中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)servletContext.getAttribute(“xxx”);，获取名为xxx的域属性；

void removeAttribute(String name)：用来移除ServletContext中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；

Enumeration getAttributeNames()：获取所有域属性的名称

|  |
| --- |
| ServletContext context = this.getServletContext(); //servletContext域对象  context.setAttribute("data", "共享数据"); //向域中存了一个data属性  在另一个servlet中，可以使用如下语句来获取域中的data属性  ServletContext context = this.getServletContext();  String value = (String) context.getAttribute("data"); //获取域中的data属性  System.out.println(value); |

### 功能1获取应用初始化参数（全局配置参数）

Servlet也可以获取初始化参数，但它是局部的参数；也就是说，一个Servlet只能获取自己的初始化参数，不能获取别人的，即初始化参数只为一个Servlet准备！

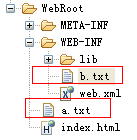
可以配置公共的初始化参数，为所有Servlet而用！这需要使用ServletContext才能使用！

还可以使用ServletContext来获取在web.xml文件中配置的应用初始化参数！注意，应用初始化参数与Servlet初始化参数不同：

web.xml

|  |
| --- |
| <web-app ...>  ...  <context-param>  <param-name>paramName1</param-name>  <param-value>paramValue1</param-value>  </context-param>  <context-param>  <param-name>paramName2</param-name>  <param-value>paramValue2</param-value>  </context-param>  </web-app> |
| ServletContext context = **this**.getServletContext();  String value1 = context.getInitParameter("paramName1");  String value2 = context.getInitParameter("paramName2");  System.*out*.println(value1 + ", " + value2);    Enumeration names = context.getInitParameterNames();  **while**(names.hasMoreElements()) {  System.*out*.println(names.nextElement());  } |

### 获取资源相关方法（Servletconfig）



1.获取真实路径（\*\*\*\*\*）

还可以使用ServletContext对象来获取Web应用下的资源，例如在hello应用的根目录下创建a.txt文件，现在想在Servlet中获取这个资源，就可以使用ServletContext来获取。

2.获取a.txt的真实路径：String realPath = servletContext.getRealPath(“/a.txt”)，realPath的值为a.txt文件的绝对路径：F:\tomcat6\webapps\hello\a.txt；

获取b.txt的真实路径：String realPath = servletContext.getRealPath(“/WEB-INF/b.txt”)；

获取资源流

不只可以获取资源的路径，还可以通过ServletContext获取资源流，即把资源以输入流的方式获取：

获取a.txt资源流：InputStream in = servletContext.getResourceAsStream(“/a.txt”)；

获取b.txt资源流：InputStream in = servletContext.getResourceAsStream(“/WEB-INF/b.txt”)；

3获取指定目录下所有资源路径

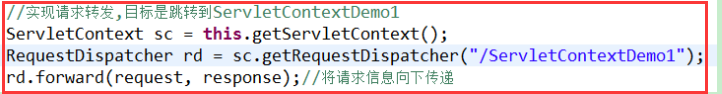
还可以使用ServletContext获取指定目录下所有资源路径，例如获取/WEB-INF下所有资源的路径：

|  |
| --- |
| Set set = context.getResourcePaths("/WEB-INF");  System.*out*.println(set); |
| [/WEB-INF/lib/, /WEB-INF/classes/, /WEB-INF/b.txt, /WEB-INF/web.xml] |

注意，本方法必须以“/”开头！！！

4实现servlet的转发(ServletContext继承了RequestDispacher)

　　RequestDispatcher getRequestDispatcher(String path) ;//参数表示要跳转到哪去



## 练习：访问量统计

一个项目中所有的资源被访问都要对访问量进行累加！

创建一个int类型的变量，用来保存访问量，然后把它保存到ServletContext的域中，这样可以保存所有的Servlet都可以访问到！

* 最初时，ServletContext中没有保存访问量相关的属性；
* 当本站第一次被访问时，创建一个变量，设置其值为1；保存到ServletContext中；
* 当以后的访问时，就可以从ServletContext中获取这个变量，然后在其基础之上加１。
* 获取ServletContext对象，查看是否存在名为count的属性，如果存在，说明不是第一次访问，如果不存在，说明是第一次访问；
* 第一次访问：调用Servletcontext的setAttribute()传递一个属性，名为count，值为1；
* 第2~N次访问：调用ServletContext的getAttribute()方法获取原来的访问量，给访问量加1，再调用Servletcontext的setAttribute()方法完成设置。

相信各位一定见过很多访问量统计的网站，即“本页面被访问过XXX次”。因为无论是哪个用户访问指定页面，都会累计访问量，所以这个访问量统计应该是整个项目共享的！很明显，这需要使用ServletContext来保存访问量。

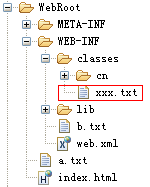
|  |
| --- |
| ServletContext application = **this**.getServletContext();  Integer count = (Integer)application.getAttribute("count");  **if**(count == **null**) {  count = 1;  } **else** {  count++;  }  response.setContentType("text/html;charset=utf-8");  response.getWriter().print("<h1>本页面一共被访问" + count + "次！</h1>");  application.setAttribute("count", count); |

## 获取类路径下资源

　　获取类路径资源，类路径对一个JavaWeb项目而言，就是/WEB-INF/classes和/WEB-INF/lib/每个jar包！

* Class
* ClassLoader：（大多情况使用它）

　　这里要讲的是获取类路径下的资源，对于JavaWeb应用而言，就是获取classes目录下的资源。



|  |
| --- |
| InputStream in = **this**.getClass().getResourceAsStream("/xxx.txt");  System.*out*.println(IOUtils.*toString*(in)); |
| InputStream in = **this**.getClass().getClassLoader().getResourceAsStream("xxx.txt");  System.*out*.println(IOUtils.*toString*(in)); |

* Class类的getResourceAsStream(String path)：
* 路径以“/”开头，相对classes路径；
* 路径不以“/”开头，相对当前class文件所有路径，例如在cn.itcast.servlet.MyServlet中执行，那么相对/classes/cn/itcast/servlet/路径；
* ClassLoader类的getResourceAsStream(String path)：
* 相对classes路径；

---------------------------------------------------------------------------------------------------------------------------------------------day09总结

1. Servlet生命周期方法是

\* init(ServletConfig)：只被调用一次

\* service(ServletRequest,ServletResponse)：被调用0~N次

\* destory()：只被调用一次

2. Http缺省请求方法是：GET

3. HttpServlet对GET请求的处理方法是：doGet()

4. <servlet>的子标签有：

\* <servlet-name>

\* <servlet-class>

\* <init-param>

\* <load-on-startup> 元素可以让服务器在启动时就创建该Servlet，其中<load-on-startup>元素的值必须是大于等于的整数，它的使用是服务器启动时创建Servlet的顺序。

<servlet>

<servlet-name>hello1</servlet-name>

<servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>

<load-on-startup>0</load-on-startup> [在<servlet>中配置<load-on-startup>，其中给出一个非负整数！]

</servlet>

<servlet-mapping>

<servlet-name>hello1</servlet-name>

<url-pattern>/hello1</url-pattern> 是<servlet-mapping>的子元素，用来指定Servlet的访问路径，即URL。它必须是以“/”开头！

</servlet-mapping>

5. ServletConfig的作用：

\* 对应<servlet>的配置信息

\* 可以获取<servlet>中配置的<init-param>信息

6. Http响应头Content-type用来说明响应数据的MIME类型

7. HttpServlet理解

\* HttpServlet是GenericServlet的子类，实现了GenericServlet中的抽象方法

\* service(ServletRequest, ServletResponse)会调用本类的service(HttpServletRequest,HttpServletResponse)

\* service(HttpServletRequest,HttpServletResponse)方法会根请求方式来调用相对的处理方法，例如请求方式为GET，那么该方法会调用doGet()方法，请求方式是POST，那么会调用doPost()方法。所以我们应该去重写HttpServlet的doGet()或doPost()方法。

8. <url-pattern>通配符配置

\* <url-pattern>中可以使用通配符

\* 通配符要么在最前面，要么在最后，例如：\*.do、/\*都是正确的，但/\*.jsp是错误的，因为\*在中间，不是在最前面，也不是在最后面。

\* 必须以“/”或“\*”开头

9. ServletContext理解

\* 一个Web应用只有一个ServletContext对象，它会在服务器启动时创建，会在服务器关闭时被销毁，它的生命与服务器相同。

\* 通常在整个Web应用中共享数据时可以使用ServletContext对象

\* 还可以使用ServletContext来获取Web资源的真实路径，servletContext.getRealPath("/WEB-INF/a.jpg");

\* ServletContext可以用来资源的MIME类型，例如：servletContext.getMimeType("a.jpg")，它会返回image/jpeg

10. <url-pattern>作用

\* 使用访问路径与Servlet绑定在一起

11. 请求转发和重定向的异同

\* 请求转发是通过RequestDispatcher对象的forward()方法完成的

\* 重定向是通过HttpServletResponse对象的sendRediect()方法完成的

\* 请求转发是在一个请求中跨越多个动态资源（jsp/servlet），所以多个动态资源之间可以共享request数据

\* 重定向是两次请求，第一次请求服务器响应给客户端的是302，以及Location响应头，通知客户端再次去请求新的资源，所以客户端又发出第二次请求。所以重定向中被请求的多个动态资源之间不能共享request数据。

\* 请求转发后，地址栏的url不会改变，因为是一个请求；

\* 重定向后，地址的rul会改变，因为是两个请求

12. 编写一个Servlet的可以通过哪些方式完成

\* 实现Servlet接口

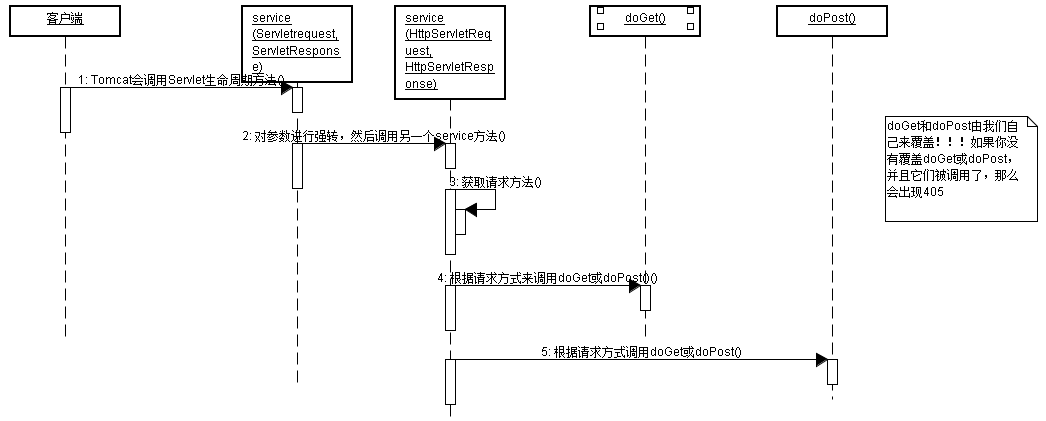
\* 继承GenericServlet类

\* 继承HttpServlet类

13. 在HttpServlet中获取ServletContext对象的方法有：

\* 通过ServletConfig类的getServletContext()方法；

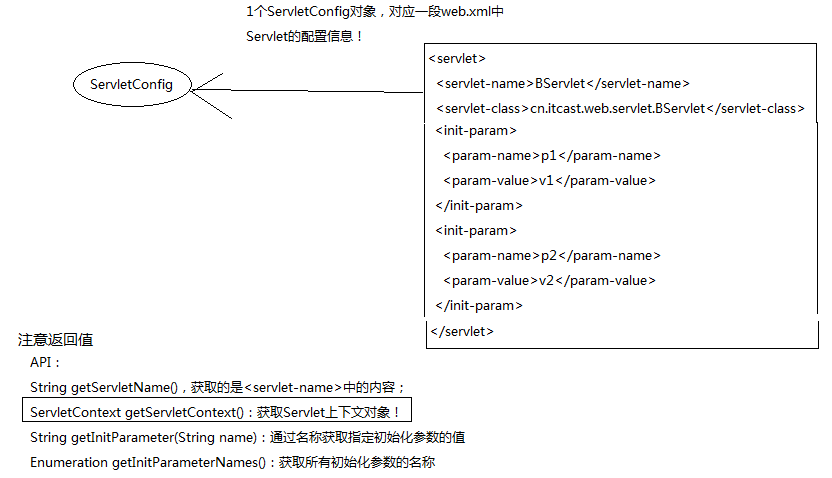
\* HttpServlet类也提供了getServletContext()方法



day09HttpServlet的时序图



day09HttpServlet的原理



day09ServletConfig是什么



day09浏览器访问Servlet的web.xml配置

--------------------------------------------------------------------------------------------------------

Servlet

1. 什么是Servlet

\* Servlet是JavaWeb三大组件之一（Servlet、Filter、Listener）

\* Servlet是用来处理客户端请求的动态资源

\* Servlet的任务有：

> 获取请求数据

> 处理请求

> 完成响应

\* Servlet接口方法：

> void init(ServletConfig)

> void service(ServletRequest,ServletResponse)

> void destory()

> ServletConfig getServletConfig()

> String getServletInfo()

2. 实现Servlet的方式

\* 实现Servlet接口（不方便）

\* 继承GenericServlet类（不方便）

\* 继承HttpServlet类（方便）

3. Servlet第一例

\* 写一个类cn.itcast.MyServlet，实现Servlet接口

\* 实现service()方法，在其中给出System.out.println("hello servlet!");

\* 在web.xml文件中指定Servlet的访问路径为：/myservlet

<servlet>

<servlet-name>xxx</servlet-name>

<servlet-class>cn.itcast.MyServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>xxx</servlet-name>

<url-pattern>/myservlet</url-pattern>

</servlet-mapping>

当用户在地址栏中访问：http://localhost:8080/day04\_1/myservlet时，会执行System.out.println("hello servlet!");

Servlet生命周期

Servlet接口一共5个方法，但其中只有三个是生命周期方法：

\* void init(ServletConfig)

\* void service(ServletRequest,ServletResponse)

\* void destory()

1). 服务器创建Servlet：

\* 当Servlet第一次被请求时，或服务器启动时，服务器会创建Servlet实例。

\* 服务器默认是在servlet第一次被请求时创建Servlet实例，如果希望服务器启动时就创建Servlet实现需要在web.xml中配置

\* 服务器只为一个类型的Servlet创建一个实例对象，所以Servlet是单例的；

2). 服务器初始化Servlet：

\* 当服务器创建Servlet实例后会马上调用Servlet的init(ServletConfig)方法，完成对Servlet的初始化；

\* init(ServletConfig)只会被调用一次

\* 服务器会在调用init()方法时传递ServletConfig参数

3). 服务器使用Servlet处理请求：

\* 当Servlet被请求时，服务器会调用Servlet的service(ServletRequest,ServletResponse)方法

\* service(ServletRequest,ServletResponse)方法每处理一次请求，就会被调用一次，所以它可能会被调用N次

\* 因为Servlet是单例的，所以可能在同一时刻一个Servlet对象会被多个请求同时访问，所以这可能出现线程案例问题

\* Servlet不是线程案例的，这有助与提高效率，但不能让Servlet具有状态，以免多个线程争抢数据

4). 服务器销毁Servlet

\* 服务器通常不会销毁Servlet，通常只有在服务器关闭时才会销毁Servlet

\* 服务器会在销毁Servlet之前调用Servlet的destory()方法

\* 可以在destory()方法中给出释放Servlet占有的资源，但通常Servlet是没什么可要释放的，所以该方法一般都是空的

ServletConfig

ServletConfig是Servlet中的init()方法的参数类型，服务器会在调用init()方法时传递ServletConfig对象给init()方法。

ServletConfig对象封装了Servlet在web.xml中的配置信息，它对应<servlet>元素。ServletConfig类的功能有：

\* String getServletName()：获取Servlet配置名，即<servlet-name>的值；

\* ServletContext getServletContext()：获取ServletContext对象，这个对象稍后介绍

\* String getInitParameter(String name)：获取初始化参数

\* Enumeration getInitParameterNames()：获取所有初始化参数的名称

在web.xml文件中，配置<servlet>时可以为<servlet>配置0~N个初始化参数，例如：

<servlet>

<servlet-name>xxx</servlet-name>

<servlet-class>cn.itcast.servlet.MyServlet</servlet-class>

<init-param>

<param-name>p1</param-name>

<param-value>v1</param-value>

</init-param>

<init-param>

<param-name>p2</param-name>

<param-value>v2</param-value>

</init-param>

</servlet>

GenericServlet

GenericServlet是Servlet接口的实现类，但它是一个抽象类，它唯一的抽象方法就是service()方法

GenericServlet实现了Servlet方法：

\* 实现了String getServletInfo()方法

\* 实现了void destory()方法，空实现

\* 实现了void init(ServletConfig)方法，用来保存ServletConfig参数

\* 实现了ServletConfig getServletConfig()方法

GenericServlet实现了ServletConfig接口：

\* 实现了ServletContext getServletContext()方法

\* 实现了String getInitParameter()方法

\* 实现了String getServletName()方法

\* 实现了Enumeration getInitParameterNames()方法

GenericServlet添加了init()方法：

\* 该方法会被init(ServletConfig)方法调用

\* 如果希望对Servlet进行初始化，那么应该覆盖init()方法，而不是init(ServletConfig)方法

HttpServlet

HttpServlet是GenericServlet的子类，它专注HTTP请求

HttpServlet类的方法：

\* 实现了void service(ServletRequest,ServletResponse)方法，实现内容是：

> 把ServletRequest强转成HttpServletRequest

> 把ServletResponse强转成HttpServletResponse

> 调用本类添加的void service(HttpServletRequest,HttpServletResponse)方法

\* 添加了void service(HttpServletRequest,HttpServletResponse)方法，内容是：

> 调用request的getMethod()获取请求方式

> 如果请求方式为GET，那么调用本类添加的doGet(HttpServletRequest,HttpServletResponse)方法

> 如果请求方式为POST，那么调用本类添加的doPost(HttpServletRequest,HttpServletResponse)方法

\* 添加了doGet(HttpServletRequest,HttpServletResponse)方法，内容是响应405，表示错误，所以我们应该去覆盖这个方法

\* 添加了doPost(HttpServletRequest,HttpServletResponse)方法，内容是响应405，表示错误，所以我们应用去覆盖这个方法

　如果是通过继承HttpServlet类来自定义Sevlet的话，那么：

\* 不要去覆盖void service(ServletRequest,ServletResponse)(注意参数)

\* 不要去覆盖void service(HttpServletRequest, HttpServletResponse)

\* 而应该去覆盖doGet()或doPost()方法。

<url-pattern>

<url-pattern>是<servlet-mapping>的子元素，用来绑定Servlet的访问路径

可以在一个<servlet-mapping>中给出多个<url-pattern>，也就是说一个Servlet可以有多个访问路径：

<servlet-mapping>

<servlet-name>xxx</servlet-name>

<url-pattern>/helo1<url-pattern>

<url-pattern>/hello2<url-pattern>

</servlet-mapping>

还可以在<url-pattern>中使用通配符，即“\*”。

\* <url-pattern>/\*<url-pattern>：表示匹配任何路径

\* <url-pattern>/do/\*<url-pattern>：表示匹配以/do开头的任何路径

\* <url-pattern>\*.do<url-pattern>：表示匹配任何以“.do”结尾的路径

注意：

\* 通配符要么在开头，要么在结尾，不能在中间，例如：/\*.do就是错误的使用。

\* 如果不使用通配符，那么<url-pattern>必须以“/”开头，例如：<url-pattern>abc</url-pattern>就是错误的

ServletContext

ServletContext是Servlet三大域对象之一

ServletContext在服务器启动时创建，在服务器关闭时销毁，一个JavaWeb应用只创建一个ServletContext对象

1. 它的功能分类：

\* 存取数据

\* 读取web.xml中的应用初始化参数

\* 读取应用资源

2. 获取ServletContext对象

在HttpServlet中可以通过以下方法来获取ServletContext对象

\* ServletContext sc = this.getServletContext()

\* ServletContext sc = this.getServletConfig().getServletContext()

2. 存取数据

因为在一个JavaWeb应用中，只有一个ServletContext对象，所以在ServletContext中保存的数据可以共整个JavaWeb应用中的动态资源共享

ServletContext是Servlet三大域对象之一，域对象内部有一个Map，用来保存数据

\* void setAttribute(String name, Object value)：用来添加或替换ServletContext域数据

> servletContext.setAttribute("xxx", "XXX")，添加域数据

> servletContext.setAttribute("xxx", "XXXX")，覆盖域数据，因为在域中已经存在了名为xxx的数据，所以这次就是覆盖了

\* Object getAttribute(String name)：通过名称来获取域数据

\* void removeAttribute(String name)：通过名称移除域数据

\* Enumeration<String> getAttributeNames()：获取所有ServletContext域数据的名称

3. 读取web.xml中配置的应用初始化参数

<context-param>

<param-name>p1</param-name>

<param-value>v1</param-value>

</context-param>

<context-param>

<param-name>p2</param-name>

<param-value>v2</param-value>

</context-param>

\* servletContext.getInitParameter("p1")，返回v1

\* servletContext.getInitParameter("p2")，返回v2

\* servletContext.getInitParameterNames()，返回Enumeration<String>，包含p1和p2

4. 获取项目资源

\* String getRealPath(String path)：获取资源的真实名称

String path = servletContext.getRealPath("/WEB-INF/a.jpg");

返回值为/WEB-INF/a.jpg真实路径，即磁盘路径：C:/tomcat6/wabapps/hello/WEB-INF/a.jpg

\* InputStream getResourceAsStream(String path)：获取资源的输入流

InputStream in = servletContext.getResourceAsStream("/WEB-INF/a.jpg");

返回的是a.jpg的输入流对象，可以从流中得到a.jpg的数据

\* Set<String> getResourcePaths(String path)：获取指定目录下的所有资源路径

Set<String> paths = servletContext.getResourcePaths("/WEB-INF");

返回的Set中包含如下字符串：

> /WEB-INF/lib/

> /WEB-INF/classes/

> /WEB-INF/web.xml

> /WEB-INF/a.jpg

获取类路径资源

可以通过Class类的对象来获取类路径下的资源，对应JavaWeb应用的类路径就是classes目录下的资源

例如：

InputStream in = cn.itcast.servlet.MyServlet.class.getResourceAsStream("a.jpg");

获取的是：/WEB-INF/classes/cn/itcast/servlet/a.jpg，即与MyServlet.class同目录下的资源

例如：

InputStream in = cn.itcast.servlet.MyServlet.class.getResourceAsStream("/a.jpg");

获取的是：/WEB-INF/classes/a.jpg，即类路径的根目录下的资源，类路径的根目录就是/classes目录

## 小点

### serlvetConfig和serlvetContext区别

servletContext接口是Servlet中最大的一个接口，呈现了web应用的Servlet视图。ServletContext实例是通过 getServletContext()方法获得的，由于HttpServlet继承Servlet的关系GenericServlet类和HttpServlet类同时具有该方法。

在这里，这个WEB应用就拥有一个它自己的仓库-ServletContext，而这10个Servlet则共享这个大仓库，并且各自拥有属于他们自己的小仓库-ServletConfig。

ServletContext就是一个全局的概念，它属于应用，但我们有时候不想让某些参数被其他Servlet应用，仅仅想在自己的Servlet中被共享，这时候就需要把它保存在ServletConfig中，换句话说，从一个Servlet来看，ServletConfig是它的全局，而从一个Servlet集合（Web应用）来看，ServletContext是它的全局。

1.Tomcat启动→读入xml文件。

2.容器为这个应用建立一个新的ServletContext实例，应用的所有部分都共享这个上下文。

3.如果xml中有定义上下文的初始参数，则容器首先创建初始参数实例（应该就像一个Bean一样）。

4.把初始化参数实例的引用交给ServletContext。

5.容器建立一个新的servlet，这时建立一个新的ServletConfig对象，并且为这个ServletConfig对象提供一个ServletContext的引用。

6.调用servlet的init（）方法初始化servlet。

servlet上下文，是针对servletconfig而提出来的，因为容器在配置文件中提取的初始化参数保存在了servletconfig对象中，但由于初始化参数只针对某个具体的servlet而言，别的servlet是访问不到这个参数的，所以为了提供一个可以供全体servlet使用的对象--这个对象也可以从配置文件中获取参数，哪个老外就弄出了一个servletcontext对象，并把它称为上下文或者应用上下文，其实就这么简单。只不过大家现在所听到的所看到的上下文被形态化了，经典话了而已。追起本质，还是很好理解的。

### ServletContext 与application的异同

相同：其实servletContext和application 是一样的，就相当于一个类创建了两个不同名称的变量。在servlet中ServletContext就是application对象。大家只要打开jsp编译过后生成的Servlet中的\_jspService()方法就可以看到如下的声明：

ServletContextapplication = null;

application= pageContext.getServletContext();

不同：两者的区别就是application用在jsp中，servletContext用在servlet中。application和page、requestsession 都是JSP中的内置对象，在后台用ServletContext存储的属性数据可以用application对象获得。而且application的作用域是整个Tomcat启动的过程。

例如:ServletContext.setAttribute("username",username);

则在JSP网页中可以使用 application.getAttribute("username");来得到这个用户名。

# JSP

## Jsp简介（java server pages）

JSP全称是 java servlet pages（服务器端页面，浏览器看不懂），它和 servlet 技术一样，都是Sun公司定义的一种用于开发动态web页面的技术JSP的网页实际上是由在HTML文件中加入的Java代码片段和JSP的特殊标记构成的。这些Java代码可以是Java Bean、SQL语句、RMI（远程方法调用）对象等。例如：一个JSP页面包含了用于产生静态网页的HTML代码，同时也包含了连接数据库的JDBC代码，那么当网页在浏览器中显示时，它既包含了静态的HTML代码，又包含了从数据库中取得的动态内容，正因为如此才能称之为动态网页。

为什么jsp也是一种动态web资源的开发技术呢？

写jsp，虽然就像是在写HTML，但是jsp允许在页面中编写java代码，并且允许开发人员在页面中获取request、response等web开发常用对象，实现与浏览器的交互，所以jsp也是一种动态web资源的开发技术；

JSP是简 Servlet编写的一种技术，它将Java代码和HTML语句混合在同一个文件中编写，只对网页中的要动态产生的内容采用Java代码来编写，而对固定不变的静态内容采用普通静态HTML页面的方式编写。

1、JSP可以放置在WEB应用程序中的除了WEB-INF及其子目录外的其他任何目录中，

JSP页面的访间路径与普通HTML页面的访间路径形式也完全一样。

2、JSP本质上是一个Servlet。WEB容器（Servlet引擎）接收到以.jsp为扩展名的URL的访问请求时，它将把该访问请求交给JSP引擎去处理。每个JSP页面在第一次被访问时，JSP引擎将它翻译成一个Servlet源程序，接着再把这个Servlet源程序编译成Servlet的class类文件，然后再由WEB容器（Serlet引擎）像调用普通Servlet程序一样的方式来装载和解

释执行这个由JSP页面翻译成的Serlet程序。

3、JSP规范也没有明确要求JSP中的脚本程序代码必须采用Java语言，JSP中的脚本程序代码可以采用Java语言之外的其他脚本语言来编写，但是JSP页面最终必须转换成Java Servlet程序。

4、JSP是简 Servlet编写的一种技术，它将Java代码和HTML语句混合在同一个文件中编写，对网页中的要动态产生的内容采用Java代码来编写，而对固定不变的静态内容采用普通静态HTML页面的方式编写。

5、jsp的本质是个servlet，servlet中的相关方法，jsp里面也可以使用。

## Jsp 调用和运行原理

浏览器访问 jsp 页面时，Web 服务器是如何调用并执行一个jsp页面的？

首先知道，我们访问服务器的资源时，无论访问的是什么？比如访问的是HTML、jsp，其实我们访问的都是一个servlet，而不是真正的jsp、html ，因此，我们访问 jsp 其实就是去访问 servlet;

Jsp在被访问的时候，服务器会将它翻译为 servlet，转换后的servlet，被保存在服务器目录下的work/项目名/apache/…

每个jsp页面在第一次被访问时，web容器都会把请求交给 jsp引擎（即一个java程序）去处理。Jsp引擎 先将 jsp 翻译成一个\_jspServlet（实际上就是一个servlet），然后按照 servlet 的调用方式进行调用 ；

由于JSP第一次访问时，会被翻译成servlet，所有第一次访问通常会比较慢，但是第二次访问，JSP引擎 如果发现 JSP 没有变化，就不再翻译，而是直接调用，所以程序的执行效率不会受到影响；

1、web 服务器在执行jsp页面时，是如何把jsp页面中的HTML排版标签输出给浏览器的？

这里其实是 jsp 对应的 servlet 的功劳，我们知道服务器会将jsp转为一个servlet 对象，我们去访问jsp的时候，实际访问到的也是这个servlet对象；在这个servlet对象对应的类中。它是通过out.writer()语句将 jsp 中的 HTML 语句，原封不动的打给浏览器；遇到<%java代码%>，就直接执行；

2、jsp页面中的java代码，服务器是如何执行的？

是原封不动的，代码最后都在在servlet类中执行了；

3、Web服务器在调用jsp时，会给jsp提供一些什么java对象？

会提供许多对象；request、response、out、application、session等这些对象，在jsp中是直接可以使用的，因为在jsp对应的servlet类中已经提供好了；

如果想要获得jsp对应的servlet类的对象的自身，使用page，也是它内置提供的。

4. sp其实是一种特殊的servle

当jsp页面第一次被访问时，服务器会把ap编译成java文件（这个java其实是一个servlet类）

>然后再把java编译成.class

>然后创建该类对象

最后调用它的service()方法

>第二次请求同一jsp时，直接调用service()方法。I

\*在tomcat的work目录下可以找到jsp对应的.java源代码。

查看sp对应java文件：

>java脚本

>html

## JSP的helloworld

在work文件夹中可以找到翻译的java文件。

新建一个Jsp页面，在body节点楼内<% %>即可编写jav代码

|  |
| --- |
| <body>  <%  Date date = **new** Date();  System.out.print(date);  %>  </body> |

## Jsp页面

|  |
| --- |
| <!--JSP指令，也是一种特殊的标签 -->  <%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>  <%  /\* java代码片段 \*/  String path = request.getContextPath();//获取项目名  /\* http://loaclhost:8080/项目名/ \*/  String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";  %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <!--向页面输出，这里会输出上面的basePath，一般不会用这个，写href时一般会写绝对路径。 -->  <base href="<%=basePath%>">    <title>My JSP 'index.jsp' starting page</title>  <meta http-equiv="pragma" content="no-cache">  <meta http-equiv="cache-control" content="no-cache">  <meta http-equiv="expires" content="0">  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">  <meta http-equiv="description" content="This is my page">  <!--  <link rel="stylesheet" type="text/css" href="styles.css">  -->  </head>    <body>  <a href ="<%=basePath%>hello.html">hello</a> <br>  <!--这里标签的相对路径相当于basePath的相对地址。 如果在文件夹下创建一个jsp，  basePath有值，相对于basePath -->  </body>  </html> |

## JSP语法：

### jsp的组成

jsp = html+java脚本+jsp标签。

模板元素：Jsp页面中的HTML内容称之为 jsp 模板元素。Jsp的模板元素定义了网页的基本骨架，即定义了页面的结构和外观。

### 注释

1、<!--html注释-->//客户端可见

2、<%--html注释--%>//客户端不可见

3、//jsp脚本注释

4、 /\*

jsp脚本多行注释

\*/

JSP引擎在将 jsp 页面翻译成 servlet 程序时，会 忽略jsp页面中被注释的内容 ；注意是被jsp引擎翻译时，就抛弃了，也就是在servlet类中将没有这个注释的内容；但是你要是使用HTML的注释，就不行了。

### jsp声明

jsp声明：定义JSP中的变量、方法以及静态方法，与在java中定义一个全局变量和共用方法时相同的。

格式：<%!变量定义/方法定义/类 %>

<jspLdeclaration>变量定义/方法定义/类</jsp:declaration> 这种方法已经过时。

注：JSP声明的结尾跟Java的结尾一样都以“；”结束，可以一次定义多个变量，利用“，”分割。声明部分的变量和方法只在当前的页面中有效。类中能放什么，里面就能放什么

|  |
| --- |
| 例如：  1、定义变量  <!% int a,b=0,c; String name=”sx” ;%>  2、定义方法  <!% int sum(int a ,in b ){  retrun a+b;  }%> |

在jsp中，脚本片段中java代码，都被翻译到servlet的jspService方法中了，那么假如，我们想要将java代码，写到jspService方法外面的话，就需要使用jsp声明了。们就可以为servlet类添加方法、字段等属性了，但是一般没啥人使用这个技术。

### 指令

JSP指合（directive)是为JSP引擎而设计的，它们并不直接产生任何可见输出，而只是告诉引拿如何处理JSP页面中的其余部分。

jsp指令的基本语法格式：

<%@指令 属性名=“值” %>

例如：<%@ page contentType="text/html; ISO-8859-1" %>

如果一个指令有多个属性，这多个属性可以写在一个指令中，用 空格 分隔多个属性，也可以分开写。

在jsp2.0规范中共定义3个指令：

page指令

Include指令

taglib指令

### jsp脚本(表达式)

语法：<%= xxxx %> 或者<jsp:expression>变量或者表达式<jsp:expression>(过时)

jsp脚本表达用于将程序数据输出到客户端，只能写在一行。Jsp引擎在翻译脚本表达式时，会将程序数据转成字符串，然后在相应的位置用out.print(...)，将数据输出给客户端。

Jsp脚本表达式的变量或者表达式后面不能有分号，其实原因很简单，脚本表达式是要被放到out.print(...)里面的，在结尾加上一个;会被输出，而我们并不想输出这个分号。

.JSP表达式（expression)提供了将一个java变量或表达式的计算结果输出到客户端的简化方式，它将要输出的变量或表达式直接封装在<%=和%>之中。

方法中可以

<%

Date date=new Date()；

out.print(date)；

%>

<%=date%>//效果一样

### Jsp脚本片段

语法：<%

多行java代码

%>

jsp脚本片段中只能出现java代码，不能出现其他模板元素，因为，jsp引擎在翻译jsp页面时，会将jsp脚本片段中的 java代码原封不动的放到 servlet 的 \_jspService 方法中。

在一个jsp页面中可以有多个脚本片段，在脚本片段之间是可以写上模板元素的；并且多个脚本片段之间的数据是共享的。单个脚本片段中的java代码可以是不完整的，但是多个脚本片段中，java代码必须完整了 。

## Page指令（与页面属性相关的指令）

page指令用于定义jsp页面的各种属性，无论page指令出现在jsp页面中的什么地方，它作用的都是整个jsp页面。为了保持程序的可读性和遵循良好的编程习惯，page指令最好是放在整个jsp页面的初始位置。

基本语法：[%@page atrribute1=’value1’ [ ...attributen= ‘valuen’]%](mailto:%25@page%20atrribute1='value1'%20%20[%20...attributen=%20'valuen']%25)

注意1、page指令有多种属性，但使用最多的是language、import、pageEncoding这3个，其中language是必须设置的，目前JSP页面使用Java 语言，所以其默认值是java；import用来声明需要导入的

包；pageEncoding属性用于设置页面的编码。在所有的属性中除import可以声明多个外，其他的属性都只能出现一次。

|  |  |
| --- | --- |
| 属性名 = “值” |  |
| pageEncoding = "xxx" | 指明翻译 jsp 的时候，使用 xxx 码表 |
| contentType = "text/html;  charset=UTF-8" | 指明jsp翻译成的 servlet 文件，里面的 response 使用的码表 |
| language = "java" | 指明 jsp 页面中的语言是 java |
| extends = "xxx.class" | 表名生成jsp翻译的servlet类继承自xxx，不过一般，继承自默认类，即可。不需要改动 |
| import = "xxx" | 表示，我们的java语句，需要导入的包，一般自动导包就好了,不用我们手动去指定这个属性 |
| session = "false / true " | false 表示，翻译过去的servlet中不要自动创建session 对象，目的是为了，在不需要 session 的情况下，避免生成 session，减轻服务器的压力。或者我们需要session对象，但是，我们想手动让服务器创建session对象的 |
| buffer = "8kb / none / xx kb" | out.print()的数据，不是直接写到浏览器中（效率太慢），还是先写到缓冲区中，再写到浏览器，buffer指定缓冲区的大小，默认是8kb |
| autoFlush = "false / true" | 是否自动刷新缓冲区，默认为true |
| isErrorPage = "true" | 表示该jsp是页面错误处理页面，属性值置为true时，jsp 翻译为 servlet时，会传进去一个异常对象，表示错误的异常；  作用：当服务器出错了，跳到错误页面（404页面），然后返回给浏览器，但这404是个正确的页面，服务器就会返回200状态码，我们为了告知浏览器我们出错了，就在页面中加上这个属性值，服务器将返回500 |
| isELIgnored = "true" | 是否忽略 EL表达式 |
| errorPage = "url" | 用来指定当前页面出错时，跳转的界面。属性值必须使用相对路径，以 / 开头则相对当前 web 应用；没有 / 打头，则相对于当前页面;  当然，我们也可以在服务器的 web.xml 文件 使用<error-page>元素为整个web应用指定错误处理页面,其中的两个子标签，一个用于指明异常的类型（写全名），这个子标签还可以写成状态码，一个用于指明出现该异常。跳准的界面地址;  如果我们在 jsp 中写明了 errorPage 属性，那么web.xml文件中的错误处理配置，对此jsp不生效 |

## Include指令

include 指令用于引入 jsp 页面，如果使用 include 引入其他 jsp 页面，那么 jsp 引擎将把这两个jsp翻译为一个servlet，当前JSP页面与静态引入的页面紧密结合为一个Servlet。

所以，include指令引入通常也称为静态导入；

include 指令是在JSP页面生成Servlet时引入需要包含的页文件，可以是HTML文件，也可以是JSP文件，还可以是其他文件（例如，js文件）。include指令的作用是在标签插入的位置插入静态的文件内容，使其与JSP文件组合成新的JSP页面，然后由JSP引擎翻译成Servlet文件，这样做有如下两个好处：

1、码可以复用，因为被引入的文件是静态文件，所以在其他的JSP页面中也可以

导入。

2、JSP页面的代码结构清晰易懂，维护也比较简单。

|  |  |
| --- | --- |
| 属性名 = “值 | 作 用 |
| <@include file = "url"> | 其中的file属性用于指定被引入文件的路径。路径以“/”开头，来表示当前web应用； |

include指令经常用来包含网站中经常出现的相同页面。例如，一般情况下，网站为每个页面都设置导航栏，把它放在页面的顶端或者左边，这部分的代码在每个页面都重复，那么就可以用include 来解决这类问题，它能够为开发者省去这些重复的动作。

### 细节：

1、被引入的文件必须遵循jsp语法

2、被引入的文件可以使用任意的扩展名，即使是HTML扩展名，jsp引擎还是会按照处理jsp页面的方式处理页面的内容；为了，见名思意，jsp规范建议使用 .jspf 作为静态导入文件的扩展名；

3、由于使用.jspf指令会涉及到两个jsp页面，并会把2个jsp翻译为一个servlet，所有这两个jsp页面的指令不能冲突；

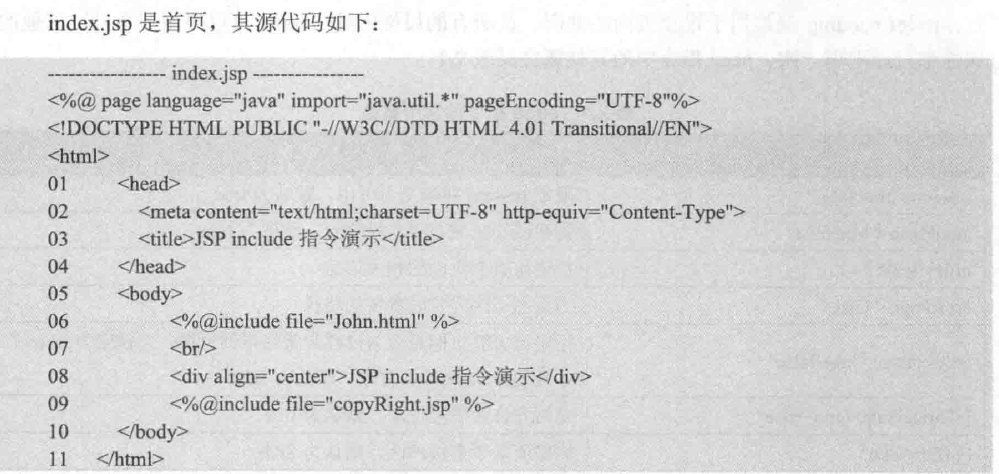
4、被导入的页面，不要包含HTML语句了，否则页面将不是一个完成的页面，会出现两个 <head> 这样的标签；

### 动态包含

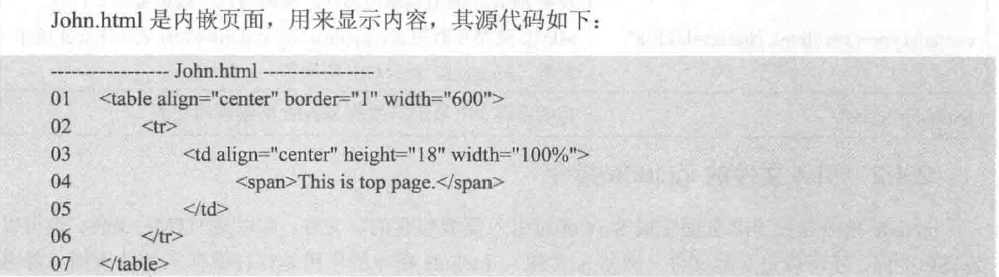
语法： request.getRequestDispatcher("url").include(request,response);

这是在运行时包含，会将每一个 jsp 都对应生成一个servlet，也就是最终生成不止一个 servlet 。我们一般选择静态导入，性能高 ；

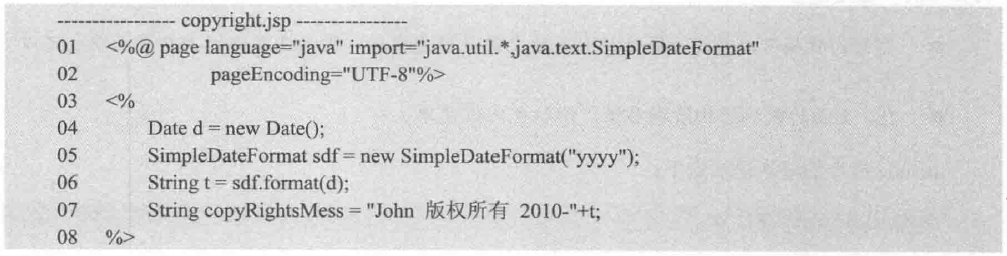
例如：



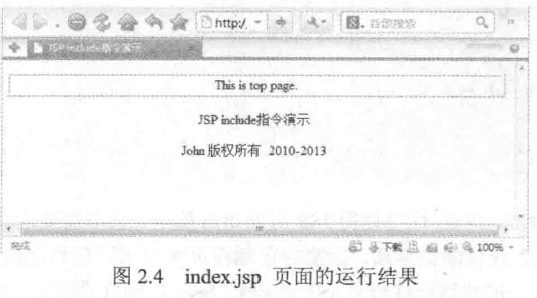
内嵌页面



版权信息



效果



## taglib指令

taglib指令（又名标签指令）是JSP新增的一个指令，用户可以自定义新的标签在页面中执行。taglib指令的语法如下：

<%taglib uri=\"tagliburl\"prefix=\"tagPre\"%>

其中uri属性用来表示自定义标签库的存放位置。prefix属性是为了区分不同标签库的标签名，在页面中引用标签也是以prefix开头的。

## page指令

page指合用于定义JSP页面的各种属性，无论page指合出现在JSP页面中的什么地方，它作用的都是整个JSP页面，为了保持程序的可读性和遵循良好的编程习惯，page指令最好是放在整个JSP页面的起始位置。

<%@page

[language="java"] //当前JSP页面使用的语言是java，但合法的语言只有java一个。

[extends="package.class"] //Servlet被编译成源文件是能够继承哪些类。（没什么用）

[import="{package.class I package.},.…"]导入包

import javax.servlet.\*; //以下为jsp默认导入的包

import javax.servlet.http.\*;

import javax.servlet.jsp.\*;

[ session="trueI false"] //session 是true或者是false表示是否能够与使用隐藏变量session

errorPage和isErrorPage

>errorPage 指定若当前页面出现错误的实际响应页面时什么。其中/表示的是当前WEB应用的根目录。

[%@page errorPage="/WEB-INF/error.jsp”%](mailto:%25@page%20prrorPage=\%22/error.jsp)

isErrorPage 指定当前页面是否为错误处理页面，可以说明当前页面是否可以使用exception隐藏变量。需要注意的是：若指定isErrorPage="true",并使用exception的方法了，一般不建议能够直接访问该页面。

在响应error.jsp时，JSP引擎使用的请求转发的方式。

>如何使客户不能直接访间某一个页面呢？对于Tomcat 服务器而言，WEB-INF下的文件是不能通过在浏览器中直接输入地址来访问的。但通过请求的转发是可以的！

contentType:指定当前JSP页面的响应类型。（可以在tomcat/conf/web.xml中查看文本类型。）

实际调用的是response.setContentType("text/html；charset=UTF-8")；

通常情况下，对于JSP页面而言其取值均为text/html；charset=UTF-8，charset返回指定的编码是什么。

pageEncoding：指定当前JSP页面的字符编码。通常情况下该值和 contentType中的 charset一致。

isELIgnored:指定当前JSP页面是否可以使用EL表达式。

## Jsp中的九大隐式对象

JSP引擎在调用JSP对应的\_JspServlet时，会传递或创建9个与WEB开发相关的对象供\_JspServlet使用。JSP技术的设计者为了方便web开发人员在编写JSP页面时，获得这些对象的引用，特意定义了9个相应的变量，开发人员在JSP页面中通过这9个变量就可以快速的获得这9大对象的引用；

这九大隐式对象就是：response、request、session、application（ServletContext）、page (jsp对象自身)、config（servlertConfig）、exception、out（JspWriter）、PageContext

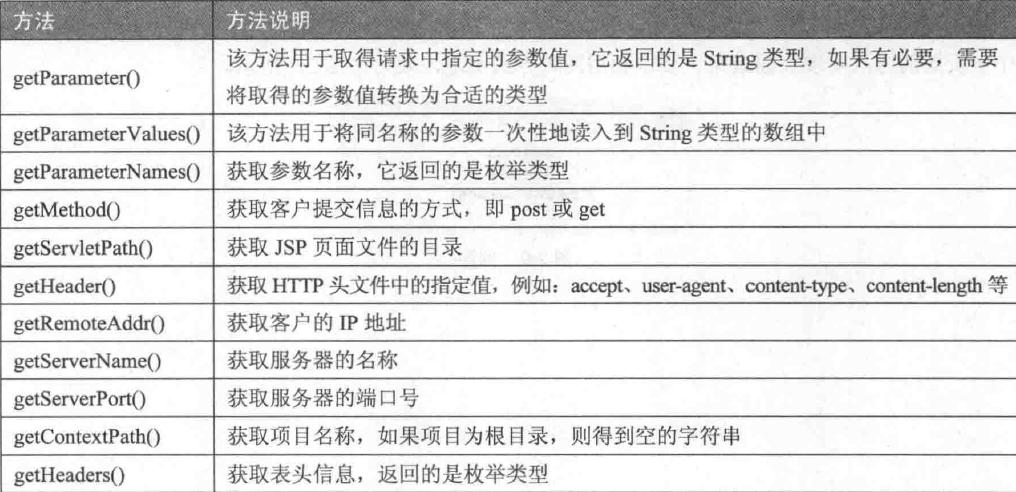
隐含对象：JSP被编译class文件中，会默认添加下列的隐含变量。.JSP页面的隐含变量：没有声明就可以使用的对象。JSP页面一共有9个隐含对象.

### jsp编译后的class文件

|  |
| --- |
| public void \_jspService(HttpServletRequest request, HttpServletResponse response)  throws java.io.IOException, ServletException {  PageContext pageContext = null;  HttpSession session = null;  ServletContext application = null;  ServletConfig config = null;  JspWriter out = null;  Object page = this;  JspWriter \_jspx\_out = null;  PageContext \_jspx\_page\_context = null;  try {  response.setContentType("text/html;charset=UTF-8");  pageContext = \_jspxFactory.getPageContext(this, request, response,  null, true, 8192, true);  \_jspx\_page\_context = pageContext;  application = pageContext.getServletContext();  config = pageContext.getServletConfig();  session = pageContext.getSession();  out = pageContext.getOut();  \_jspx\_out = out;  　　　　从这里开始，才是JSP页面的内容  }… |

### request

本节将要介绍request对象的作用范围及其常用的方法。用户每访问一个页面，就会产生一个HTTP 请求。这些请求中一般都包含了请求所需的参数值或者信息，如果将 request对象看做是客户请求的一个实例，那么这个实例就包含了客户请求的所有数据，因此，可以通过request来获取客户端和服务器端的信息，如IP地址、传递的参数名和参数值、应用系统名称、服务器主机名称等。



### response

本节将介绍response对象的常用方法。当用户访问一个页面时，就会产生一个HTTP请求，服务器做出响应时调用的是response响应包。response响应包实现的是接口javax.servlet.http.HtpServletResponse。



设置头信息：设置头信息包括设置页面返回的MIME类型、返回的字符集、页面中的meta等信息。其中设置MIME类型和返回的字符集尤为重要，因为它关系到页面的显示是否会出现乱码。有两种方法用于设置MIME类型和返回的字符集，分别如下。

response.setContentType(“text/html;charset=utf-8”) ;

<%@page contentType=”text/html;charset=utf-8 language=’’java”%>

设置页面重定向

response.sendRedirect(String url);url:可以是相对路径，也可以是绝对路径。

### pageContext

在JavaWeb中一共四个域对象，其中Servlet中可以使用的是request、session、application三个对象，而在JSP中可以使用pageContext、request、session、application四个域对象。

pageContext是jsp技术中最重要的一个对象，它代表jsp页面的运行环境生命周期是一个jsp页面。这个对象自身就封装可其他8大隐式对象的引用。它自身还是一个域对象，可以用来保存数据；这个对象中，还封装了web开发中经常涉及到的一些常用操作，例如引入和跳转其他资源、检索其他域对象中的属性；

功能:域对象功能、代理其它域对象功能、获取其他内置对象；

1、通过PageContext对象获得其他对象PageContext对象中包含其他8大隐式对象的引用。

通过 getXXX()方法；

首先知道一个事实：JSP页面中是不应该出现java代码，良好的jsp中，是不应该出现一行java代码的。但是有些时候，我们为了获取servlet传过来的数据，不得已要使用java代码，这时候，我们就需要一个技术：自定义标签；（以后会讲这个技术）

我们把java代码替换成一个自定义标签，这个标签对应着一个java类，我们需要把web中的这些对象传给java类，一个一个传，很麻烦的，因此，我们就直接传一个pageContext对象过去；

2、功能: pageContext也是域对象，它的范围是当前页面。它的范围也是四个域对象中最小的！

 void setAttribute(String name, Object value);

 Object getAttrbiute(String name, Object value);

 void removeAttribute(String name, Object value);

3、代理其它域对象功能

还可以使用pageContext来代理其它3个域对象的功能，也就是说可以使用pageContext向request、session、application对象中存取数据，例如：

|  |
| --- |
| pageContext.setAttribute("x", "X");  pageContext.setAttribute("x", "XX", PageContext.REQUEST\_SCOPE);  pageContext.setAttribute("x", "XXX", PageContext.SESSION\_SCOPE);  pageContext.setAttribute("x", "XXXX", PageContext.APPLICATION\_SCOPE); |

### session

4、session:代表浏览器和服务器的一次会话，是HttpSession的一个对象。后面详细学习。常用

### application

application 对象实现的接口为javax.servlet.ServletContext，它的生命周期是从application 对象创建到应用服务器关闭，也就是说当服务器关闭时application对象才消失。可以将它视为Web应用的全局变量，当服务器运行时有效，如果关闭服务器，其中保存的信息也就消失了。



### config

6、config：当前JSP对应的 Servlet的 Servletconfig 对象（几乎不使用）。若需要获取JSP配置的初始化参数，需要通过映射的地址才可以。几乎不用

Web.xml中的代码：

<servlet>

<servlet-name>hellojsp</servlet-name>

<jsp-file>/hel1o.jsp</jsp-file>

<init-param>

<param-name>test</param-name>

<param-value>testValue</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>hellojsp</servlet-name>

<ur1-pattern>/hellojsp</url-pattern>

</servlet-mapping>

### out

作用：Out隐式对象 用于向客户端发送文本数据;

Out对象是通过调用 PageContext 对象的getOut()方法返回的，其作用和用法与servletResponse.getWriter方法返回的 PrintWriter 对象非常相似。

JSP页面中的out隐式对象的类型为JspWriter，JspWriter相当于一种带缓存功能的PrintWriter，设置JSP页面的page指令的buffer属性可以调整它的缓存大小，甚至关闭它的缓存(buffer属性的值为none)；

只有向out对象中写入了内容，且 满足如下任何一个条件时(相当于刷新缓冲区)，out对象才会去调用servletResponse.getWriter方法，并且通过该方法返回PrinterWriter对象将out对象缓冲区中的内容 真正的写入 到servlet引擎提供的缓冲区中（response缓冲区）。

条件：

√ 设置page指令的buffer属性关闭了out对象的缓存功能

√ out对象的缓冲区已满 ；

√ 整个JSP页面结束；

备注：

Out对象输出的数据 可能滞后于 response.getWriter 对象输出的数据，原因就在于out对象自己的缓冲区；out对象的缓存区满了，才将数据刷新到response的缓冲区；服务器发现response缓冲区有数据，才会将数据发送到客户端；

### page

1. page:只想当前JSP

### exception

1. exception：特殊的情况下使用。

<% page isErrorPage=”true”%> 指定为true是才能够使用。

pageContex t,request, session ,appocation(对属性的作用于范围从小到大)

## jsp标签

JSP还提供了一种称之为Action的元素，在JSP页面中使用Action元秦可以完成各种通用的JSP页面功能，也可以实现一些处理复杂业务逻辑的专用功能。

1、<isp:include>标签用于把另外一个资源的输出内容插入进当前JSP页面的输出内容之中，这种在JSP页面执行时的引入方式称之为动态引入。

Action元素采用XML元素的语法格式，即每个Action元素在JSP页面中都以XML标签的形式出现。JSP规范中定义了一些标准的Action元素，这些元素的标签名都以isp作为前缀，并且全部采用小写，例如，<p:include>、<jsp:forward>等等。

1、.jsp:incluce标签：

<jsp:include page="b.jsp"><jsp:include>

2）.动态引入：并不是像include 指令生成一个Servlet 源文件，而是生成两个Servlet源文件，然后通过一个方法的方式把目标页面包含进来。

org.apache.jasper.runtime.JspRuntimelibrary.include(request,response,\"b.jsp\",out,false);

10、中文乱码问题

1、1）.在JSP页面上输入中文，请求页面后不出现乱码：保证 contentType="text/html；charset=UTF-8"pageEncoding="UTF-8"，charset和pageEcoding的编码一致，且都支持中文，通常建议取值UTF-8。还需要保证浏览器的显示的字符编码也和请求的JSP页面编码一致。

2、获取中文参数值

2）.获取中文参数值：默认参数在传输过程中使用的编码为ISO-8859-1

对于POST请求：只要在获取请求信息之前，调用request.setcharacterEncoding（\"UTF-8\")即可。（在调用request.getParameter 或者是request.getReader等）

对于GET请求：前面的方式对于GET无效。可以通过修改Tomcat的 server.xml文件的方式。

参照http://1ocalhost:8989/docs/config/index.html文档的 useBodyEncodingForURI 属性。

为Connector节点添加 useBodyEncodingForURI="true"属性即可。

## javaweb的四个域

|  |  |
| --- | --- |
| 域 | 范围（从小到大） |
| PageContext域 | pageContext，最小的域，只能在页面中使用 |
| Request域 | 请求域 |
| Session域 | 会话级别的域 |
| application域： | 最大的域，在整个应用程序中可用 |

用于查找某个属性的属性值，它会依次从page、request、session、application四个域，从小到大的查找，在某个域中查到数据，即刻返回这个数据，不会再继续查下去。如果四个域都没查到，则返回null；

这个方法，便利于我们可以在jsp中直接使用这个方法来查找数据，而不要去关注数据在哪一个域中；

### PageContext域的方法

1、获取数据

pageContext.getAttribute(String) ;

2、设置数据

pageContext.setAttribute(String,String);

3、移除数据

pageContext.removeAttribute(String);

### PageContext跨域访问的方法

1、获取id对应的域的数据

pageContext.getAttribute(String，id) ;

2、设置id对应的域数据

ageContext.setAttribute(String,String,id);

3、移除id对应的域数据

pageContext.removeAttribute(String,id);

4、PageContext查找属性的方法

pageContext.findAttribute(String)

## jsp的基本语法

3、JSP的脚本片段

JSP脚本片断（scriptlet）是指嵌套在<%和%>之中的一条或多条Java程序代码。在JSP脚本片断中，可以定义变量、执行基本的程序运算、调用其他Java类、访问数据库、访问文件系统等普通Java程序所能实现的功能。在JSP脚本片断可以直接使用JSP提供的隐式对象来完成WEB应用程序特有的功能。JSP脚本片断中的Java代码将被原封不动地搬移进由JSP页面所翻译成的Serlet的ispService方法中，所以，JSP脚本片断之中只能是符合Java语法要求的程序代码，除此之外的任何文本、HTML标记、其他JSP元素都必须在脚本片断之外编写。JSP脚本片断中的Java代码必须严格遵循Java语法，例如，每条命执行语句后面必须用分号（；）结束。在一个JSP页面中可以有多个脚本片断（每个脚本片断代码嵌套在各自独立的一对<%和%>之间），在两个或多个脚本片断之间可以嵌入文本、HTML标记和其他JSP元素。

<%

String ageStr=request.getParameter("age")；

Integer age=Integer.parseInt(ageStr);

if(age>=18){

%>

成人...

<%

}else{

%>

未成人...

<%

4、JSP声明（JSP代码中基本不这样使用）

JSP声明：JSP声明将Java代码封装在<%！和%>之中，它里面的代码将被插入进Servlet的\_jspService方法的外面。

5、Jsp的注释

<%-- JSP注释--> <!— Html 的注释-->

区别：Jsp的注释可以阻止java代码的执行。

## 属性相关的方法

Object getAttribute(String name):获取指定的属性

Enumeration getAttributeNames():获取所有的属性的名字组成的Enumeration对象

removeAttribute(String name):移除指定的属性

void setAttribute(String name,Object o):设置属性

## 引入和跳转到其它的资源

PageContext类中定义了一个forward方法和两个include方法来分别简化和替代RequestDispatcher(...).forward方法和RequestDispatcher(...).include方法。、

### 原始操作

|  |
| --- |
| // forward  request.getRequestDispatcher("/xxx").forward(request, response);  // include  request.getRequestDispatcher("/xxx").include(request, response); |

## 乱码问题

### 乱码的原因

1. jsp文件保存的编码和服务器翻译servlet的时候的编码不一致。

2. 浏览器的编码和翻译的servlet的 response 的编码编码不一致。

### 解决方法

解决方法：<%@ page pageEncoding="utf-8" contentType="text/html;charset=UTF-8" %>

我们发现只要这两个地方使用相同的编码，乱码就不会产生。

pageEncoding=”xxx” 告诉服务器翻译时使用什么码表；

contentType="text/html;charset=UTF-8" 再告诉浏览器用什么码表打开 ；

其实可以不告诉浏览器用什么码表，因为只要告诉服务器翻译的码表，服务器会自动将servlet的response的码表设为一致的；但是，我们还是写上吧。

## jsp的动作

在JSP中一共有13个动作：<jsp:include>、<jsp:forward>、<jsp:plugin>、

<jsp:param>、<jsp:useBean>、<jsp:getProperty>、<jsp:setProperty>、<jsp:output>、<jsp:attribute>、<jsp:element>、<jsp:body>、<jsp:params>、<jsp:falback>。

其中<jsp:include>、<jsp:forward>、<jsp:param>使用较多，下面将进行主要介绍。

### <jsp:include>

格式<jsp:include page="relative URL" flush="true">

例如：<jsp:include page="top.html" flush="true">

# 域对象

## 1、简介

## 2、方法

void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如： servletContext.setAttribute(“xxx”, “XXX”)，在 ServletContext 中保存了一个域属性， 域属性名称为 xxx，域属性的值为 XXX。请注意，如果多次调用该方法， 并且使用相同的name ， 那么会覆盖上一次的值，这一特性与Map 相同；

Object getAttribute(String name) ：用来获取 ServletContext 中的数据， 当前在获取之前需要先 去存储才行，例如： String value =(String)servletContext.getAttribute( “xxx”);，获取名为 xxx 的 域属性；

void removeAttribute(String name)：用来移除 ServletContext中的域属性，如果参数name 指 定的域属性不存在，那么本方法什么都不做；

Enumeration getAttributeNames()：获取所有域属性的名称

pagecontext, request, session, application 对象都有这些方法，四个对象为域对象。

pageContext:属性的作用范围仅限于当前JSP页面

request:属性的作用范围仅限于同一个请求。

session:属性的作用范围限于一次会话：浏览器打开直到关闭称之为一次会话（在此期间会话不失效）

Application：属性的作用范围限于当前的web应用

# Cookie

## 什么是会话跟踪技术

我们需要先了解一下什么是会话！可以把会话理解为客户端与服务器之间的一次会晤，在一次会晤中可能会包含多次请求和响应。例如你给10086打个电话，你就是客户端，而10086服务人员就是服务器了。从双方接通电话那一刻起，会话就开始了，到某一方挂断电话表示会话结束。在通话过程中，你会向10086发出多个请求，那么这多个请求都在一个会话中。

在JavaWeb中，客户向某一服务器发出第一个请求开始，会话就开始了，直到客户关闭了浏览器会话结束。

在一个会话的多个请求中共享数据，这就是会话跟踪技术。例如在一个会话中的请求如下：

1.请求银行主页；

2.请求登录（请求参数是用户名和密码）；

3.请求转账（请求参数与转账相关的数据）；

4.请求信誉卡还款（请求参数与还款相关的数据）。

会话路径技术使用Cookie或session完成

我们知道HTTP协议是无状态协议，也就是说每个请求都是独立的！无法记录前一次请求的状态。但HTTP协议中可以使用Cookie来完成会话跟踪！

在JavaWeb中，使用session来完成会话跟踪，session底层依赖Cookie技术。

在人机交互过程中，会话管理是指保持用户的整个会话活动的交互与计算机系统跟踪的过程。会话管理分为：桌面会话管理、浏览器会话管理、Web会话管理。本书讨论的是Web会话管理（通常指的是Session 以及Cookie)，也称为会话跟踪。

## 会话技术的三种方式

三种解决方案：使用隐藏域、使用Cookie、使用URL重写。

### 使用隐藏域

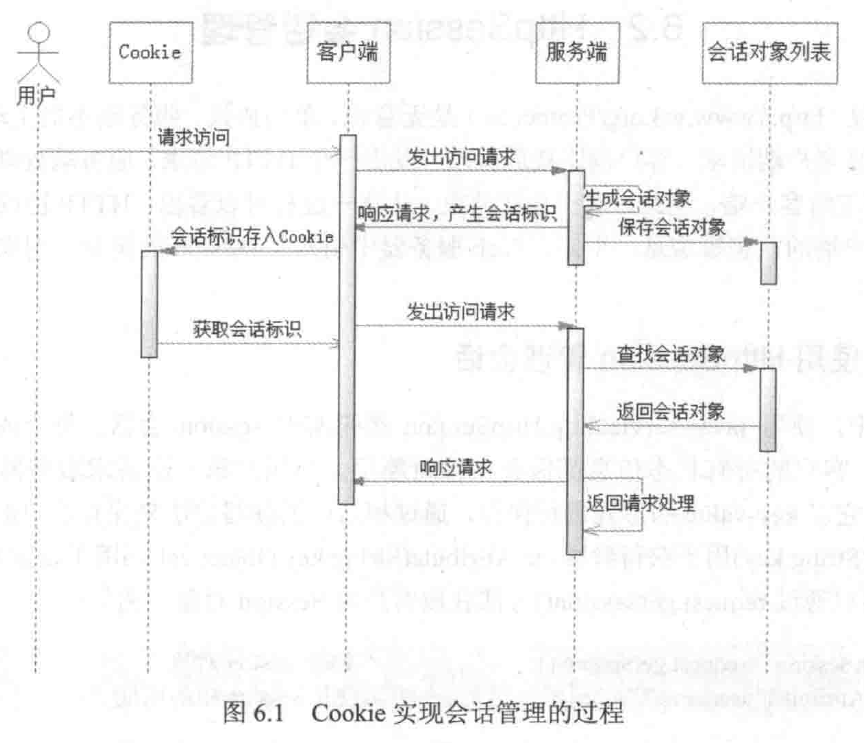
隐藏域是指将表单中的内容，在显示页面时隐藏，即不显示数据。在JSP中将input标签的type属性值设定为hidden，即生成一个隐藏表单域。再将会话的唯一标识记录到隐藏域中的value属性中，并设定name属性值。当表单提交时，会话标识也被提交到服务端，服务端根据它找到对应的会话对象。

使用隐藏域时，需要在每个页面中都包含会话标识表单，这样才能在许多页面跳转之间保存会话，并对它进行管理。此方法实现起来比较繁琐，在目前的开发中较少使用。

### 使用Cookie

利用Cookie实现会话管理是最容易的实现方式，也是使用较多的方法。其原理是在服务端保存的会话对象中设定会话的唯一标识，客户端将会话标识保存在Cookie中，当浏览器发起请求时，从Cookie中取得会话标识并发送给服务端，服务端在收到请求后，根据发送过来的会话标识查找到对应的会话对象，这样服务端就清楚当前是哪个客户端在连接，并且可以从会话中获得信息。

利用Cookie实现会话管理是目前开发中采用的主流方法，大多数的动态页面开发技术都实现了这一功能，并且其管理流程是自动完成的，在实现上并不需要多大的开发工作量。



### 6.1.3使用URL重写

顾名思义，URL重写是指在URL地址的末尾添加会话标识，改写了原先的URL地址。其本质是用于唯一标识会话的信息，以参数的形式添加到URL中，服务端接收到请求时，解析出会话标识，然后利用会话标识查找出与当前请求对应的会话对象。该方法是用在浏览器中Cookie被禁用的情况下，利用该方法可以很好地实现会话跟踪而不会受到浏览器参数设定的影响。

利用URL重写的方法，其特点是在URL后面添加会话标识，因此整个Web应用中的超链接或者脚本中用到的URL都需要添加会话标识，Web应用中的每一个页面都需要动态生成，页面中的每一个链接或者由客户端生成的跳转指令都必须加上会话标识，这样才能确保是当前会话。当客户端访问静态页面时，会话标识将会丢失，当重回动态页面时将不能继续此前的会话，以上是它的特点，也是它的缺陷。

利用该方法还有一个问题，即当用户在浏览网页时，可能会将URL复制下来分享给朋友，因为URL中会包含会话标识，那么其他人可能与当前浏览者使用同一会话对象，这样当前浏览者的个人隐私就暴露了，信息不再安全。无论采用哪种方式实现会话管理，在服务端都需要完成相关代码才能完整地实现会话管理的任务。这些任务主要是：生成唯一的会话标识、存储会话对象、从Web容器中取得当前请求的会话、回收空闲会

## 什么叫Cookie？

Cookie翻译成中文是小甜点，小饼干的意思。在HTTP中它表示服务器送给客户端浏览器的小甜点。其实Cookie就是一个键和一个值构成的，随着服务器端的响应发送给客户端浏览器。然后客户端浏览器会把Cookie保存起来，当下一次再访问服务器时把Cookie再发送给服务器。

Cookie是由服务器创建，然后通过响应发送给客户端的一个键值对。客户端会保存Cookie，并会标注出Cookie的来源（哪个服务器的Cookie）。当客户端向服务器发出请求时会把所有这个服务器Cookie包含在请求中发送给服务器，这样服务器就可以识别客户端了！

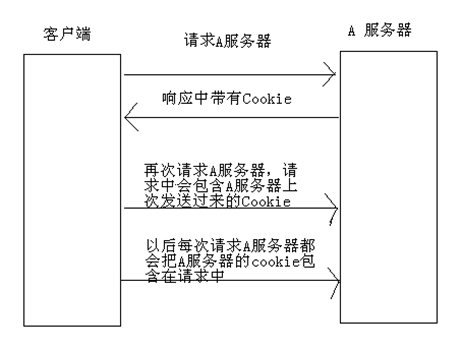
服务器保存的响应头：Set-Cookie：aaa=AAA

Set-Cookie： bbb=BBB

通过设置响应头来设置Cookie：response.addHeader(“Set-Cookie”,”aaa=AAA”)

浏览器归还Cookie：aaa=AAA;BBB=bbb;

主要作用：发送的不同请求的数据共享问题。



### Cookie的生命

什么是Cookie的生命？

Cookie不只是有name和value，Cookie还是生命。所谓生命就是Cookie在客户端的有效时间，可以通过setMaxAge(int)来设置Cookie的有效时间，存储在客户端的硬盘中，有效期内符合路径的要求都会附带该信息。。

cookie.setMaxAge(-1)：cookie的maxAge属性的默认值就是-1，表示只在浏览器内存中存活。一旦关闭浏览器窗口，那么cookie就会消失。

cookie.setMaxAge(60\*60)：表示cookie对象可存活1小时。当生命大于0时，浏览器会把Cookie保存到硬盘上，就算关闭浏览器，就算重启客户端电脑，cookie也会存活1小时；

cookie.setMaxAge(0)：cookie生命等于0是一个特殊的值，它表示cookie被作废！也就是说，如果原来浏览器已经保存了这个Cookie，那么可以通过Cookie的setMaxAge(0)来删除这个Cookie。无论是在浏览器内存中，还是在客户端硬盘上都会删除这个Cookie。

## Cookie规范

1、Cookie大小上限为4KB；

2、一个服务器最多在客户端浏览器上保存20个Cookie；

3、一个浏览器最多保存300个Cookie；

4、cookie是与浏览器绑定的，不同浏览器存储cookie的位置不同，且cookie在不同浏览器之间不能共享。

5、cookie不支持中文。

6、cookie不能存储对象，只能是字符串。

上面的数据只是HTTP的Cookie规范，但在浏览器大战的今天，一些浏览器为了打败对手，为了展现自己的能力起见，可能对Cookie规范“扩展”了一些，例如每个Cookie的大小为8KB，最多可保存500个Cookie等！但也不会出现把你硬盘占满的可能！

注意，不同浏览器之间是不共享Cookie的。也就是说在你使用IE访问服务器时，服务器会把Cookie发给IE，然后由IE保存起来，当你在使用FireFox访问服务器时，不可能把IE保存的Cookie发送给服务器。

### Cookie的覆盖

如果服务器端发送重复的Cookie那么会覆盖原有的Cookie，例如客户端的第一个请求服务器端发送的Cookie是：Set-Cookie: a=A；第二请求服务器端发送的是：Set-Cookie: a=AA，那么客户端只留下一个Cookie，即：a=AA。

### Cookie与HTTP头

Cookie是通过HTTP请求和响应头在客户端和服务器端传递的：

1、Cookie：请求头，客户端发送给服务器端；

格式：Cookie: a=A; b=B; c=C。即多个Cookie用分号离开；

2、Set-Cookie：响应头，服务器端发送给客户端；

一个Cookie对象一个Set-Cookie：

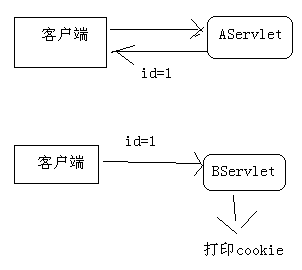
Set-Cookie: a=A

Set-Cookie: b=B

Set-Cookie: c=C;

## Cookie第一例

我们这个案例是，客户端访问AServlet，AServlet在响应中添加Cookie，浏览器会自动保存Cookie。然后客户端访问BServlet，这时浏览器会自动在请求中带上Cookie，BServlet获取请求中的Cookie打印出来。实际上，添加Cookie实际上把Cookie添加到了请求头中。



|  |
| --- |
| Cookie的创建和存储  1、创建Cookie对象  Cookie c=new Cookie(String name,String value);//cookie默认不能存储中文  //设置cookie（可选）  2、设置有效期  c.setMaxAge(int seconds)；  3、设置有效路径  c.setPath(String uri)  4、响应Cookie信息给客户端  resp.addCookie(c)；  /\*\*  \* 给客户端发送Cookie  \* **@author** Administrator  \*  \*/  **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.setContentType("text/html;charset=utf-8");    String id = UUID.*randomUUID*().toString();//生成一个随机字符串  //1、创建Cookie对象，指定名字和值  Cookie cookie = **new** Cookie("id", id);  Cookie cookie1 = **new** Cookie("id1", id1);  //2、设置Cookie的有效期  cookie.setMaxAge(3\*24\*3600);//三天  cookie1.setMaxAge(2\*24\*3600);//两天  //2、在响应中添加Cookie对象  response.addCookie(cookie);  response.getWriter().print("已经给你发送了ID");  }  } |

BServlet.java

|  |
| --- |
| **public** **class** BServlet **extends** HttpServlet {  **public** **void** doGet(HttpServle tRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.setContentType("text/html;charset=utf-8");  1、获取所有的Cookie  Cookie[] cs = request.getCookies();//获取请求头中的Cookie  **if**(cs != **null**) {//如果请求中存在Cookie，头信息中可能没有Cookie，所以要对Cookie进行判断是否为null；  **for**(Cookie c : cs) {//遍历所有Cookie  //使用getName()方法获cookie中的属性名。  **if**(c.getName().equals("id")) {//获取Cookie名字，如果Cookie名字是id，打印Cookie值  //使用getValue()获取id的属性值。  response.getWriter().print("您的ID是：" + c.getValue());//  }  }  }  }  }  //返回值  Set-Cookie：id=id;  Set-Cookie：id1=id1; |

## EL表达式获取cookie

首先cookie是EL表达式的内置对象可以直接拿来使用${cookie.name}将获得对应cookie的对象，比如我们用jsp将一段cookie发送给客户端。

Cookie cookie = new Cookie("username", "Username in cookie");

response.addCookie(cookie);

创建一个名称为username，值为"Username in cookie"的Cookie对象，然后发送给客户端。

然后我们就可以使用${cookie.username}获得这个cookie了，${cookie.username.name}获得cookie名称--"username"，${cookie.username.value}获得cookie值--"Username in cookie"。

## Cookie的path

### 什么是Cookie的路径

首先：Cookie的路径不是Cookie的保存路径。

现在有WEB应用A，向客户端发送了10个Cookie，这就说明客户端无论访问应用A的哪个Servlet都会把这10个Cookie包含在请求中！但是也许只有AServlet需要读取请求中的Cookie，而其他Servlet根本就不会获取请求中的Cookie。这说明客户端浏览器有时发送这些Cookie是多余的！

可以通过设置Cookie的path来指定浏览器，在访问什么样的路径时，包含什么样的Cookie。

### 设置Cookie的路径

Cookie提供了setPath() 方法去设置Cookie的有效路径，有如下几种情况：

1.cookie.setPath("\")：表示tomcat 服务器上的所有web项目都可以访问当前Cookie

2.cookie.setPath（"/day5")：表示day5项目下的所有web资源都可以访问当前Cookie

3.cookie.setPath("/day5/demo"):表示day5项目下demo 路径下所有 web资源都可以访问当前Cookie。但"/day5"或"/day5/aaa”将不能访问

注意：

1.Cookie 默认有效路径是创建该Cookie的web资源所在的路径，通常就是整个web项目，所以Aserylet 创建的 Cookie 能被BServlet访问。

2.如果两个Cookie的Path和name 都一样，两次调用addCookie（），后者会覆盖前者。

3.动态设置Cookie的path

cookie.setPath(request.getContextPath());

4.Jsessionid的路径为整个项目名。

### Cookie路径与请求路径的关系

下面我们来看看Cookie路径的作用：

下面是客户端浏览器保存的3个Cookie的路径：

a:　/cookietest；

b:　/cookietest/servlet；

c:　/cookietest/jsp；

下面是浏览器请求的URL：

A:　http://localhost:8080/cookietest/AServlet；

B:　http://localhost:8080/cookietest/servlet/BServlet；

C:　<http://localhost:8080/cookietest/jsp/CServlet>

 请求A时，会在请求中包含a；

 请求B时，会在请求中包含a、b；

 请求C时，会在请求中包含a、c；

也就是说，请求路径如果包含了Cookie路径，那么会在请求中包含这个Cookie，否则不会请求中不会包含这个Cookie。

 A请求的URL包含了“/cookietest”，所以会在请求中包含路径为“/cookietest”的Cookie；

 B请求的URL包含了“/cookietest”，以及“/cookietest/servlet”，所以请求中包含路径为“/cookietest”和“/cookietest/servlet”两个Cookie；

 B请求的URL包含了“/cookietest”，以及“/cookietest/jsp”，所以请求中包含路径为“/cookietest”和“/cookietest/jsp”两个Cookie；

### 设置Cookie的路径

设置Cookie的路径需要使用setPath()方法，例如：cookie.setPath(“/cookietest/servlet”);

如果没有设置Cookie的路径，那么Cookie路径的默认值当前访问资源所在路径，例如：

 访问http://localhost:8080/cookietest/AServlet时添加的Cookie默认路径为/cookietest；

 访问http://localhost:8080/cookietest/servlet/BServlet时添加的Cookie默认路径为/cookietest/servlet；

 访问http://localhost:8080/cookietest/jsp/BServlet时添加的Cookie默认路径为/cookietest/jsp；

### Cookie的domain

Cookie的domain属性可以让网站中二级域共享Cookie，次要！

百度你是了解的对吧！

http://www.baidu.com

http://zhidao.baidu.com

http://news.baidu.com

<http://tieba.baidu.com>

现在我希望在这些主机之间共享Cookie（例如在www.baidu.com中响应的cookie，可以在news.baidu.com请求中包含）。很明显，现在不是路径的问题了，而是主机的问题，即域名的问题。处理这一问题其实很简单，只需要下面两步：

 设置Cookie的path为“/”：c.setPath(“/”)；

 设置Cookie的domain为“.baidu.com”：c.setDomain(“.baidu.com”)

当domain为“.baidu.com”时，无论前缀是什么，都会共享Cookie的。但是现在我们需要设置两个虚拟主机：www.baidu.com和news.baidu.com。

第一步：设置windows的DNS路径解析

找到C:\WINDOWS\system32\drivers\etc\hosts文件，添加如下内容

|  |
| --- |
| 127.0.0.1 localhost  127.0.0.1 www.baidu.com  127.0.0.1 news.baidu.com |

第二步：设置Tomcat虚拟主机

找到server.xml文件，添加<Host>元素，内容如下：

|  |
| --- |
| <Host name="www.baidu.com" appBase="F:\webapps\www"  unpackWARs="true" autoDeploy="true"  xmlValidation="false" xmlNamespaceAware="false"/>  <Host name="news.baidu.com" appBase="F:\webapps\news"  unpackWARs="true" autoDeploy="true"  xmlValidation="false" xmlNamespaceAware="false"/> |

第三步：创建A项目，创建AServlet，设置Cookie。

|  |
| --- |
| Cookie c = **new** Cookie("id", "baidu");  c.setPath("/");  c.setDomain(".baidu.com");  c.setMaxAge(60\*60);  response.addCookie(c);  response.getWriter().print("OK"); |

把A项目的WebRoot目录复制到F:\webapps\www目录下，并把WebRoot目录的名字修改为ROOT。

第四步：创建B项目，创建BServlet，获取Cookie，并打印出来。

|  |
| --- |
| Cookie[] cs = request.getCookies();  **if**(cs != **null**) {  **for**(Cookie c : cs) {  String s = c.getName() + ": " + c.getValue() + "<br/>";  response.getWriter().print(s);  }  } |

把B项目的WebRoot目录复制到F:\webapps\news目录下，并把WebRoot目录的名字修改为ROOT。

第五步：访问www.baidu.com\AServlet，然后再访问news.baidu.com\BServlet。

### Cookie中保存中文

Cookie的name和value都不能使用中文，如果希望在Cookie中使用中文，那么需要先对中文进行URL编码，然后把编码后的字符串放到Cookie中。

　向客户端响应中添加Cookie

|  |
| --- |
| String name = URLEncoder.*encode*("姓名", "UTF-8");  String value = URLEncoder.*encode*("张三", "UTF-8");  Cookie c = **new** Cookie(name, value);  c.setMaxAge(3600);  response.addCookie(c); |

从客户端请求中获取Cookie

|  |
| --- |
| response.setContentType("text/html;charset=utf-8");  Cookie[] cs = request.getCookies();  **if**(cs != **null**) {  **for**(Cookie c : cs) {  String name = URLDecoder.*decode*(c.getName(), "UTF-8");  String value = URLDecoder.*decode*(c.getValue(), "UTF-8");  String s = name + ": " + value + "<br/>";  response.getWriter().print(s);  }  } |

# session

## HttpSession概述

### Session引入

问题：重定向中，一个用户的不同请求处理的数据共享怎么办？

解决：使用session技术。不同请求拿到相同的对象。每一个用户有一个session。

原理：用户第一次访问服务器，服务器会创建一个session对象给此用户，并将该seesion对象的sessionid使用Cookie技术存储到浏览器中，保证用户的其他请求能够获取到统一个session对象，也保证了不同请求能够获取到共享的数据。

HTTP协议（http:/www.w3.org/Protocols/)是无意识、单向协议。服务端不能主动连接客户端，只能等待并答复客户端请求。客户端连接服务端，发出一个HTTP请求，服务端处理请求，并返回一个HTTP响应给客户端，至此，本次会话结束，从这一过程可以看出，HTTP协议本身并不支持服务端保存客户端的状态等信息。于是，Web服务器中引入了session的概念，用来保存客户端的信息。

注：session与http协议不相关， 只在javaweb中存在。但是cookie是与http协议相关的，他会保存在头中。

### 什么是HttpSesssion

j获取HttpSession对象

在Java中，使用javax.servlet.http.HpSession类来实现session会话。每个请求者对应一个Session对象，客户的所有状态信息都保存在该对象里。当用户第一次请求服务器时，就创建了Session对象。它以key-value的形式进行保存，通过相对应的读写方法来保存客户的状态信息，例如getAttribute（String key)用于获得参数，setAttribute（String key,Object value)用于设定参数和参数值。

在Servlet中可以通过request.getSession0方法获取客户的Session对象，例如：

HttpSession session =request.getSession();/获取Session对象

Session.setAtribute("username","John'");/设置Session中的属性

在JSP中内置了Session对象，可以直接使用，Servlet使用上述方法获取Session对象。若JSP页面中设定了<%@page session="false"%>，则该JSP页面的Session对象是不可用的。

### HttpSession是域对象

我们已经学习过HttpServletRequest、ServletContext，它们都是域对象，现在我们又学习了一个HttpSession，它也是域对象。它们三个是Servlet中可以使用的域对象，而JSP中可以多使用一个域对象，明天我们再讲解JSP的第四个域对象。

 HttpServletRequest：一个请求创建一个request对象，所以在同一个请求中可以共享request，例如一个请求从AServlet转发到BServlet，那么AServlet和BServlet可以共享request域中的数据；

 ServletContext(application)：一个应用只创建一个ServletContext对象，所以在ServletContext中的数据可以在整个应用中共享，只要不启动服务器，那么ServletContext中的数据就可以共享；

 HttpSession：一个会话创建一个HttpSession对象，同一会话中的多个请求中可以共享session中的数据,一个会话与一个用户有关。一个session很少出现并发请求，即一个一个session对应两个用户。

下列是session的域方法：

void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：session.setAttribute(“xxx”, “XXX”)，在session中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；

 Object getAttribute(String name)：用来获取session中的数据，当前在获取之前需要先去存储才行，例如：String value = (String) session.getAttribute(“xxx”);，获取名为xxx的域属性；

 void removeAttribute(String name)：用来移除HttpSession中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；

 Enumeration getAttributeNames()：获取所有域属性的名称；

## 登录案例

需要的页面：

login.jsp：登录页面，提供登录表单；

 index1.jsp：主页，显示当前用户名称，如果没有登录，显示您还没登录；

 index2.jsp：主页，显示当前用户名称，如果没有登录，显示您还没登录；

Servlet：

LoginServlet：在login.jsp页面提交表单时，请求本Servlet。在本Servlet中获取用户名、密码进行校验，如果用户名、密码错误，显示“用户名或密码错误”，如果正确保存用户名session中，然后重定向到index1.jsp；

当用户没有登录时访问index1.jsp或index2.jsp，显示“您还没有登录”。如果用户在login.jsp登录成功后到达index1.jsp页面会显示当前用户名，而且不用再次登录去访问index2.jsp也会显示用户名。因为多次请求在一个会话范围，index1.jsp和index2.jsp都会到session中获取用户名，session对象在一个会话中是相同的，所以都可以获取到用户名。

login.jsp

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>login.jsp</title>  </head>    <body>  <h1>login.jsp</h1>  <hr/>  <form action=*"/day06\_4/LoginServlet"* method=*"post"*>  用户名：<input type=*"text"* name=*"username"* /><br/>  <input type=*"submit"* value=*"Submit"*/>  </form>  </body>  </html> |

index1.jsp

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>index1.jsp</title>  </head>    <body>  <h1>index1.jsp</h1>  <%  String username = (String)session.getAttribute("username");  **if**(username == **null**) {  out.print("您还没有登录！");  } **else** {  out.print("用户名：" + username);  }  %>  <hr/>  <a href=*"/day06\_4/index2.jsp"*>index2</a>  </body>  </html> |

index2.jsp

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>index2.jsp</title>  </head>    <body>  <h1>index2.jsp</h1>  <%  String username = (String)session.getAttribute("username");  **if**(username == **null**) {  out.print("您还没有登录！");  } **else** {  out.print("用户名：" + username);  }  %>  <hr/>  <a href=*"/day06\_4/index1.jsp"*>index1</a>  </body>  </html> |

LoginServlet

|  |
| --- |
| **public** **class** LoginServlet **extends** HttpServlet {  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setCharacterEncoding("utf-8");  response.setContentType("text/html;charset=utf-8");    String username = request.getParameter("username");      **if**(username.equalsIgnoreCase("itcast")) {  response.getWriter().print("用户名或密码错误！");  } **else** {  1、创建或获取session对象  HttpSession hs = request.getSession();  //1、创建session对象。  2、将cookie对象创建。  3、将cookie信息响应。  System.out.println(hs.getId())；//获取session的id。  2、存储数据setAttribute(String name, Object value)，值可以为任何对象。  String name = “张三”；  hs.setAttribute(“name”，name);  设置sesison存储时间,当访问的间隔超过5s中，那么服务器会重新创建一个session  注：在指定的时间内session对象没有被使用则被销毁，如果使用了则重新记时。  hs.setMaxInactiveInterval(5);  session.setAttribute("username", username);  response.sendRedirect("/day06\_4/index1.jsp");  //如果是第一次访问，则request.getSession();创建一个session，如果是第二次请求则直接返回上一个session对象。  如果请求中拥有session的标识符也就是JSESSIONID，则返回其对应的session队形。如果请求中没有session的标识符也就是JSESSIONID，则创建新的session对象，并将其JSESSIONID作为从cookie数据存储到浏览器中。  如果session对象失效了，也会创建一个session对象，将其JSESSIONID存储在浏览器的内存中。  设置sesison强制失效 强制退出。  hs.invalidate();  }  }  } |

|  |
| --- |
| /\*\*  \* session技术学习:  \* 问题：  \* 一个用户的不同请求处理的数据共享怎么办？  \* 解决：  \* 使用session技术  \* 原理：  \* 用户第一次访问服务器，服务器会创建一个session对象给此用户，并将  \* 该session对象的JSESSIONID使用Cookie技术存储到浏览器中，保证  \* 用户的其他请求能够获取到同一个session对象，也保证了不同请求能够获取到  \* 共享的数据。  \* 特点：  \* 存储在服务器端  \* 服务器进行创建  \* 依赖Cookie技术  \* 一次会话  \* 默认存储时间是30分钟  \* 作用：  \* 解决了一个用户不同请求处理的数据共享问题  \* 使用：  \* 创建session对象/获取session对象  HttpSession hs=req.getSession();  如果请求中拥有session的标识符也就是JSESSIONID，则返回其对应的session对象。  如果请求中没有session的标识符也就是JSESSIONID，则创建新的session对象，并将其JSESSIONID作为从cookie数据存储到浏览器内存中  \* 如果session对象是失效了，也会重新创建一个session对象，并将其JSESSIONID存储在浏览器内存中。  \* 设置session存储时间  \* hs.setMaxInactiveInterval(int seconds);  \* 注意：  \* 在指定的时间内session对象没有被使用则销毁，如果使用了则重新计时。  \* 设置session强制失效，session对象会被删出。  \* hs.invalidate();  \* 存储和获取数据  \* 存储：hs.setAttribute(String name,Object value);  \* 获取：hs.getAttribute(String name) 返回的数据类型为Object  \* 注意：  \* 存储的动作和取出的动作发生在不同的请求中，但是存储要先于取出执行。  \* 使用时机:  \* 一般用户在登陆web项目时会将用户的个人信息存储到Sesion中，供该用户的其他请求使用。  \* 总结：  \* session解决了一个用户的不同请求的数据共享问题，只要在JSESSIONID不失效和session对象不失效的情况下。  \* 用户的任意请求在处理时都能获取到同一个session对象。  \* 作用域：  \* 一次会话  \* 在JSESSIONID和SESSION对象不失效的情况下为整个项目内。  \* session失效处理：  \* 将用户请求中的JSESSIONID和后台获取到的SESSION对象的JSESSIONID进行比对，如果一致  \* 则session没有失效，如果不一致则证明session失效了。重定向到登录页面，让用户重新登录。  \* 注意：  \* JSESSIONID存储在了Cookie的临时存储空间中，浏览器关闭即失效。 |

## session的实现原理

session底层是依赖Cookie的！我们来理解一下session的原理吧！

session是服务器端的存储技术，由服务器创建，cookie是浏览器端的存储技术。

当我首次去银行时，因为还没有账号，所以需要开一个账号，我获得的是银行卡，而银行这边的数据库中留下了我的账号，我的钱是保存在银行的账号中，而我带走的是我的卡号。

当我再次去银行时，只需要带上我的卡，而无需再次开一个账号了。只要带上我的卡，那么我在银行操作的一定是我的账号！

当首次使用session时，服务器端要创建session，session是保存在服务器端，而给客户端的session的id（一个cookie中保存了sessionId）。客户端带走的是sessionId，而数据是保存在session中。

当客户端再次访问服务器时，在请求中会带上sessionId，而服务器会通过sessionId找到对应的session，而无需再创建新的session。



## session与浏览器

session保存在服务器，而sessionId通过Cookie发送给客户端，但这个Cookie的生命不-1，即只在浏览器内存中存在，也就是说如果用户关闭了浏览器，那么这个Cookie就丢失了。

当用户再次打开浏览器访问服务器时，就不会有sessionId发送给服务器，那么服务器会认为你没有session，所以服务器会创建一个新的session，并在响应中把sessionId中到Cookie中发送给客户端，称为一次会话。

你可能会说，那原来的session对象会怎样？当一个session长时间没人使用的话，服务器会把session删除了！这个时长在Tomcat中配置是30分钟，可以在${CATALANA}/conf/web.xml找到这个配置，当然你也可以在自己的web.xml中覆盖这个配置！在tomcat中配置的是全部的，在项目配置的是局部的。

web.xml

|  |
| --- |
| <session-config>  <session-timeout>30</session-timeout>  </session-config> |

　session失效时间也说明一个问题！如果你打开网站的一个页面开始长时间不动，超出了30分钟后，再去点击链接或提交表单时你会发现，你的session已经丢失了！

## URL重写

我们知道session依赖Cookie，那么session为什么依赖Cookie呢？因为服务器需要在每次请求中获取sessionId，然后找到客户端的session对象。那么如果客户端浏览器关闭了Cookie呢？那么session是不是就会不存在了呢？

其实还有一种方法让服务器收到的每个请求中都带有sessioinId，那就是URL重写！在每个页面中的每个链接和表单中都添加名为jSessionId的参数，值为当前sessionid。当用户点击链接或提交表单时也服务器可以通过获取jSessionId这个参数来得到客户端的sessionId，找到sessoin对象。

index.jsp

|  |
| --- |
| <body>  <h1>URL重写</h1>  <a href=*'/day06\_5/index.jsp;jsessionid=*<%=session.getId() %>*'* >主页</a>  <form action=*'/day06\_5/index.jsp;jsessionid=*<%=session.getId() %>*'* method=*"post"*>  <input type=*"submit"* value=*"提交"*/>  </form>  </body> |

也可以使用response.encodeURL()对每个请求的URL处理，这个方法会自动追加jsessionid参数，与上面我们手动添加是一样的效果。

|  |
| --- |
| <a href=*'*<%=response.encodeURL("/day06\_5/index.jsp") %>*'* >主页</a>  <form action=*'*<%=response.encodeURL("/day06\_5/index.jsp") %>*'* method=*"post"*>  <input type=*"submit"* value=*"提交"*/>  </form> |

使用response.encodeURL()更加“智能”，它会判断客户端浏览器是否禁用了Cookie，如果禁用了，那么这个方法在URL后面追加jsessionid，否则不会追加。

## 案例：一次性图片验证

# javaweb的MVC设计模式

## JAVAEE的开发思想

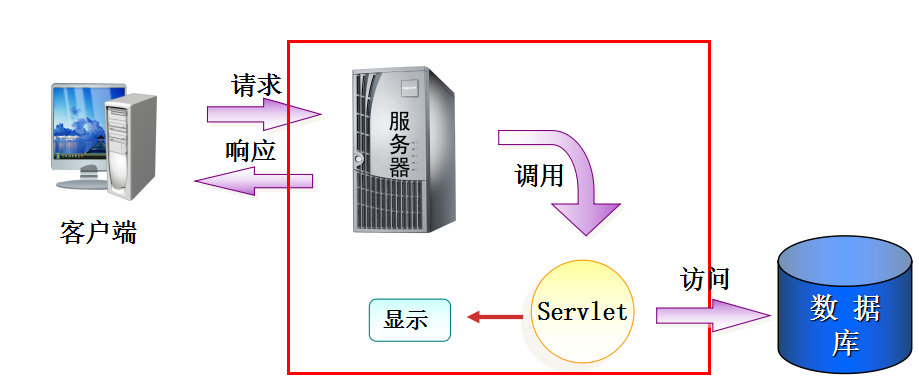


## JAVAEE的开发流程

### sevlet

1）Servlet方式：

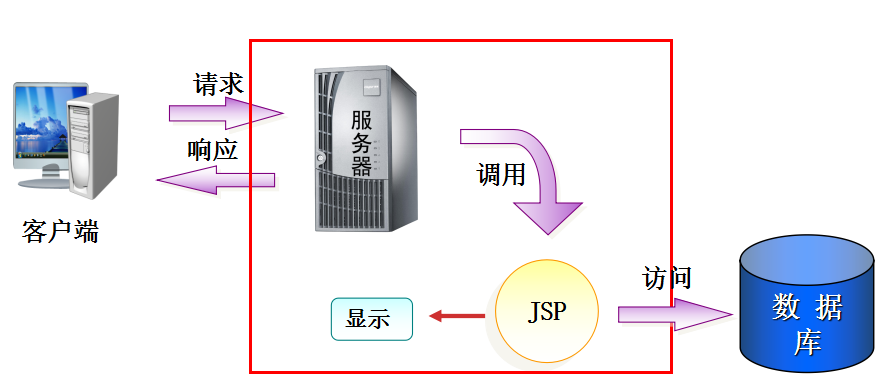
　　　　缺点：① 显示麻烦，servlet不擅长显示  ② 操作数据库的代码不能重用



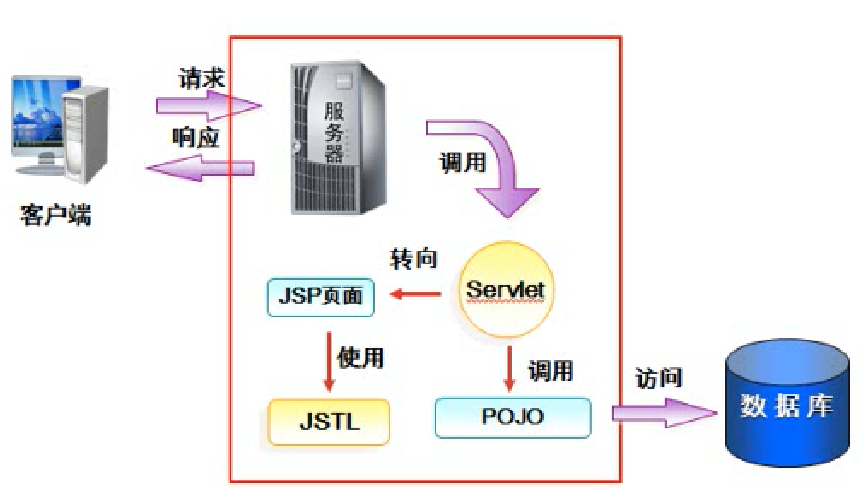
### 2、jsp

优点：友好的显示

缺点：jsp访问数据库代码不能重用，jsp中的存在大量的java代码（逻辑，访问数据库），不便于维护和升级



JSP Model是JavaWeb早期的模型，它适合小型Web项目，开发成本低！Model1第一代时期，服务器端只有JSP页面，所有的操作都在JSP页面中，连访问数据库的API也在JSP页面中完成。也就是说，所有的东西都耦合在一起，对后期的维护和扩展极为不利。



## dao层设计模式

DAO全称是（Data Access Object，数据库访问对象），主要功能就是用于进行数据操作的，在程序的标准开发架构中属于数据层的操作。

显示层：主要使用JSP/Servlet进行页面效果的显示

业务层：（Business Object，数据对象）会将多个原子性的DAO操作进行组合，组合成一个完整的业务逻辑。

数据层：（DAO，Data Access Object，数据库访问对象）提供多个原子性的DAO操作，例如：增、删、改、查，都是原子性操作。

## MVC的概念

MVC是Model-View-Controller的简称，即模型-视图-控制器。MVC是一种设计模式，它把应用程序分成三个核心模块：模型、视图、控制器，它们各自处理自己的任务。

### model（POJO和数据库）

模型是应用程序的主体部分，模型表示业务数据和业务逻辑。访问数据库的部分是model，还有一些逻辑部分，比如，购买商品后实现的商品增减操作。

模型能为多个视图提供数据。可以为不同系统提供收数据，如安卓、ios等。由于应用于模型的代码只需写一次就可以被多个视图重用，所以提高了代码的可重用性。

1.M:Model.Dao

2.V:View.JSP,在页面上填写Java代码实现显示

3.C：Controller.Serlvet:受理请求，获取请求参数，调用DAO方法，转发（或重定向）页面。

步骤：

1.受理请求

2.获取请求参数

3.调用DAO方法

4.可能会把DAO方法的返回值放入request中

5.转发（或重定向）页面

### 视图

视图是用户看到并与之交互的界面，作用如下：

1.用户显示相关的数据。

2.输入。

3.任何实际的业务处理。请求发给model，由model处理。

### 控制器

控制器接受用户的输入并调用模型和视图去完成用户的需求。

控制器接收请求并决定调用哪个模型组件去处理请求，然后决定调用哪个视图来显示模型处理返回的数据。

### 运行流程：

运行流程：客户端发送请求到服务器，服务器调Servlet(控制器)， 接收请求， 根据请求的情况，决定调用哪个类的哪个方法。

　POJO包含处理逻辑，业务逻辑，访问数据库。得到处理结果。将结果返回到Servlet中。

　 servet根据返回的结果，转向不同的jsp页面（转发方式，重定向方式。

### 详细说明

　I. Model(模型):（数据库 + POJO）普通的java类，和访问数据库

　　　　　　　>模型是应用程序的主体部分，模型表示业务数据和业务逻辑;

　　　　　　　>一个模型能为多个视图提供数据;

　　　　　　 >由于应用于模型的代码只需写一次就可以被多个视图重用，所以提高了代码的可重用性。

　　　　　II. View(视图):（jsp页面，标准的标签库 ,EL, struts2 ,spingMVC）

　　　　　　 >视图是用户看到并与之交互的界面;

　　　　　　　>视图向用户显示相关的数据;

　　　　　　　>接受用户的输入;

　　　　　　　>不进行任何实际的业务处理.

　　　　　III. Controller(控制)：（servlet）

　　　　　　 >控制器接受用户的输入并调用模型和视图去完成用户的需求;

　　　　　　 >控制器接收请求并决定调用哪个模型组件去处理请求，然后决定调用哪个视图来显示模型处理返回的数据。

### 什么时候转发，什么时候重定向？

若目标的响应页面不需要从request中读取任何值，则可以使用重定向。（还可以防止表单的重复提交）。

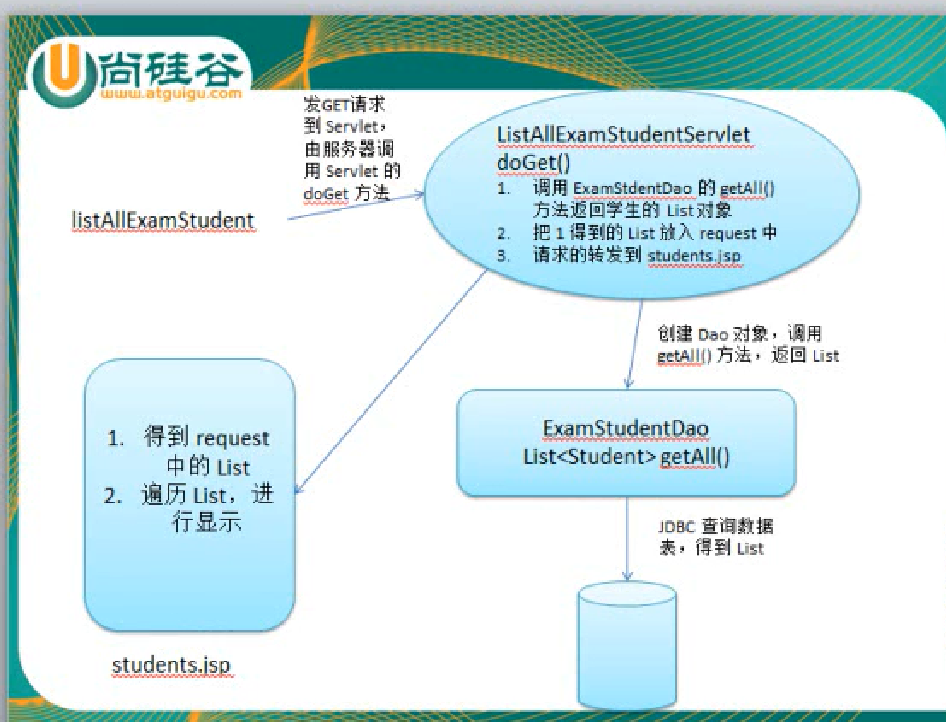
### 3.不足：

1.使用数据库连接池，DBUtils，JDBCUtis工具类，DAO基类

2.一个请求一个Serlvet不好！一个模块使用一个Sertvet，即多个请求可

以使用一个Servlet

3.在页面上加入iQuery提示



## 案例

|  |
| --- |
| MVC案例  -没有业务层，直接由Servlet调用DAO，所以也没有  事务操作。所以可以在DAO中接获取Connection  对象  -采取MVC设计模式  -使用到的技术：  ·MVC设计模式：JSP、Servlet，POJO  ·数据库使用MySQL  ·连接数据库需要使用C3P0数据库连接池  ·JDBC工具采用DBUtils  ·页面上的提示操作使用iQuery  难点：  1、多个请求如何使用一个Servlet？  2、模糊查询、  3、在创建或修改的情况下，验证用户名是否已经被占用，并给出提示。  架构：  View：JSP  1、呈现数据  2、接收用户的输入  3、编写js代码给出对应的提示、  Controller：Servlet  1、获取请求信息：获取请求参数。  2、验证请求参数的合法性,验证失败，需要返回页面，并给出提示信息。  3、把请求参数封装成一个javaBean  4、调用dao的方法  5、把返回的结果放到request中  5、响应页面：转发、重定向：如果方法request中。  View：JSP  1、呈现数据  2、接收用户的输入  3、编写js代码给出对应的提示  Model：Dao  1、获取数据库连接  2、执行CRUD  3、返回  步骤：  Mysql：创建数据表，加入数据  编写DOA,JdbcUtils工具类和CustomerDao接口 |

# EL（表达式语言）

## EL（express language）概述

### EL

主要目的：在JSP页面中，经常利用JSP表达式<%=变量或者表达式%>来输出声明的变量以及页面传递的参数，当变量很多的时候，书写这样的表达式会显得累赘，EL标签很好地解决了这个问题，它简化了表达式。

JSP2.0要把html和css分离、要把html和javascript分离、要把Java脚本替换成标签。标签的好处是非Java人员都可以使用。

JSP2.0 – 纯标签页面，即：不包含<% … %>、<%! … %>，以及<%= … %>

EL（Expression Language）是一门表达式语言，它对应<%=…%>。我们知道在JSP中，表达式会被输出，所以EL表达式也会被输出。

### EL的作用

1.获取数据：EL表达式主要用于替换jsp页面中的脚本表达式，以从各种类型的web域中检索java对象，获取数据。（某个web域中的对象，访问javaBean的属性、访问list集合，访问map集合、访问数组。

2. 执行运算：利用EL表达式可以在jsp页面中执行一些基本的关系运算、逻辑运算和算数运算，以在jsp页面中完成一些简单的逻辑运算。${user==null}

3. 获取web开发常用对象：EL表达式定义了一些隐式对象，利用这些隐式对象，web开发人员可以很轻松获得对web常用对象的引用，从而获得这些对象中的数据。

4. 调用java方法：EL表达式允许用户开发自定义EL函数，以在jsp页面中通过EL表达式调用java类的方法。

### EL的格式

格式：${参数名}

例如：${param.name}

作用：获取参数的name值，等同于<=request.getParameter(“name”)%>

### EL表达式的启用和关闭

页面禁止：如果希望整个JSP忽略EL表达式，需要在page指令中指定isELIgnored=”true”。

EL的转义：如果希望忽略某个EL表达式，可以在EL表达式之前添加“\”，例如：\${1 + 2}。

### EL不显示null

　　当EL表达式的值为null时，会在页面上显示空白，即什么都不显示。

### EL和Jsp的区别

### [ ]与.运算符

EL 提供"."和"[ ]"两种运算符来存取数据。当要存取的属性名称中包含一些特殊字符，如.或?等并非字母或数字的符号，就一定要使用"[ ]"。

例如：${user.My-Name} 应当改为${user["My-Name"]}

${user.My-Name} 应当改为${user["My-Name"]}

${sessionScope.user[data]}中data 是一个变量

### 变量

EL存取变量数据的方法很简单，例如：${username}。它的意思是取出某一范围中名称为username的变量。因为我们并没有指定哪一个范围的username，所以它会依序从Page、Request、Session、Application范围查找。假如途中找到username，就直接回传，不再继续找下去，但是假如全部的范围都没有找到时，就回传null。

## EL表达式的功能

1、操作List和数组：${list[0]}、${arr[0]}。

2、操作bean的属性：${person.name}、${person[‘name’]}，对应person.getName()方法。

3、操作Map的值：${map.key}、${map[‘key’]}，对应map.get(key)。

通过EL提供的”[]”和”.”运算符可以访问数据。通常情况下这两个运算符是等价的，可以相互代替。但也不是所有情况下都可以相互代替，例如当对象的属性名中包含一些特殊的符号(-或.或数字时)时，就只能使用[]来访问对喜爱那个的属性。例如${userInfo[user-id]}是正确的，而${userInfo.user-name}是错误的。

## 访问集合元素

### 获取数组元素(只能使用[]，不能用.)

|  |
| --- |
| 数组或集合的索引只能使用[]如：  假设数组为基本类型数据  Char[] char={“1”.”2”};  request.setAttribute(“char”,char);  ${char[1]} //获取的是值  数组中的对象  User[] users=new User[10]；  for(int i=0;i<10;i++)  {  User u=new User();  u.setUsername(“U\_”+i);  users[i]=u;  }  request.setAttribute(“users”,users);  ${users[1].username} //获取的是user对象。通过user.[num]回去其中的get方法 |

### List集合元素的获取(只能使用[]，不能用.)

|  |
| --- |
| <%  User user1 = new User(2, "李四", "女");  User user2 = new User(3, "王五", "男");  List<User> userList = new ArrayList<User>();  userList.add(user1);  userList.add(user2);  //把List集合存储到requeset域中  request.setAttribute("users", userList);  %>  <!-- 获取List集合中第2个元素 :语法：${集合的key[索引]} -->  第2个用户的名字： ${users[1].username}<br> |

### map集合元素的获取

|  |
| --- |
| <%  Map<String, String> myMap = new HashMap<String, String>();  myMap.put("a", "1111");  myMap.put("b", "2222");  myMap.put("1", "3333");  //把Map集合存到request域中  request.setAttribute("map", myMap);  %>  <!-- 获取Map集合中第2个元素的value :语法：${集合的key.元素的key},  如果元素的key是数字(以数字开头或全数字),语法：${集合的key["元素的key"]}  当然，不管元素的key是否是数字，都可以用[]语法去访问对应的value  -->  第1个元素的value: ${map['a']}<br>  第2个元素的value: ${map.b }<br>  第3个元素的value: ${map["1"] } |

## EL运算符操作

|  |  |  |  |
| --- | --- | --- | --- |
| 算术运算符 | 说明 | 范例 | 结果 |
| + | 加 | ${17+5} | 22 |
| - | 减 | ${17-5} | 12 |
| \* | 乘 | ${17\*5} | 85 |
| /或div | 除 | ${17/5}或${17 div 5} | 3 |
| %或mod | 取余 | ${17%5}或${17 mod 5} | 2 |
| 逻辑运算符 | 说明 | 范例 | 结果 |
| ==或eq | 等于 | ${5==5}或${5 eq 5} | true |
| !=或ne | 不等于 | ${5!=5}或${5 ne 5} | false |
| <或lt | 小于 | ${3<5}或${3 lt 5} | true |
| >或gt | 大于 | ${3>5}或${3 gt 5} | false |
| <=或le | 小于等于 | ${3<=5}或${3 le 5} | true |
| >=或ge | 大于等于 | ${3>=5}或${3 ge 5} | false |
| &&或and | 并且 | ${true&&false}或${true and false} | false |
| !或not | 非 | ${!true}或${not true} | false |
| ||或or | 或者 | ${true||false}或${true or false} | true |
| 特殊运算符 | 说明 | 范例 | 结果 |
| ${条件表达式？ 表达式1 ： 表达式2} |  |  |  |
| empty | 是否为空 | ${empty “”}，可以判断字符串、数据、集合的长度是否为0，为0返回true。empty还可以与not或!一起使用。${not empty “”}  也可以判断对象的是否为null或者为空串，如果为null或者空串，则返回true。 | true |
| ()运算符 |  |  |  |
|  |  |  |  |

三元运算

|  |
| --- |
|  |

符

empty演示

|  |
| --- |
| 1)判断一个变量是否为null 。  2判断一个字符串是否是空串。  3)判断一个集合是否有元素。  <%request.setAttribute("user" , "");%>  <%request.setAttribute("user1" , null);%>  <%request.setAttribute("user2" , 2);%>    <%  request.setAttribute(“num”, 5);  request.setAttribute(“str”, “”);  //创建一个集合  ArrayList list = new ArrayList();  pageContext.setAttribute(“list”, list);  %>  1) 创建一个变量是否为null  ${empty num}  2) 创建一个字符串是否是空串  ${empty str}  3) 创建一个集合是否有元素  ${empty list } |

## 访问JavaBean中的属性

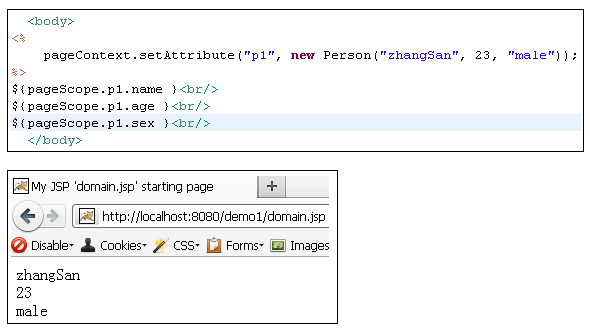
如果在域中保存的是JavaBean对象，那么可以使用EL来访问JavaBean属性。因为EL只做读取操作，所以JavaBean一定要提供get方法，而set方法没有要求。这里通过调用对象.属性的方法调用，实际上是调用的是getter()方法。

注：1、也可以调用给你自己创建的get方法，例如定义getInfor方法，可以使用$(person.info)表达式调用，要将get之后的部分进行小写来调用，并且JavaBean中不需要此属性。

2、如果Person类中的属性包含一个Animal对象，可以通过${person. animalName.name}方式调用Animal类中getName()方法。

Person.java

|  |
| --- |
| **public** **class** Person {  **private** String name;  **private** **int** age;  **private** String sex;  private Animal animalName  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** String getSex() {  **return** sex;  }  **public** **void** setSex(String sex) {  **this**.sex = sex;  }  //调用Javabean中属性不存在的方法  **public** int getInfo() {  **return** age+1;  }  } |



## 访问jsp中不同域的对象

在Web中有4个作用域分别是：page、request、session、和application。EL标签可以对这4个作用域的参数进行访问。四个域中查找相应的对象，找到则返回相应对象，找不到则返回””（注意，不是null，而是空字符串）

作用域对象可以通过xxxScopeEL的隐式变量或获取参数值。也可以直接通过参数名的方式获取。

1、pageScope：${pageScope.name}等同与pageContext.getAttribute(“name”)；

2、requestScope：${requestScope.name}等同与request.getAttribute(“name”)；

3、sessionScoep： ${sessionScope.name}等同与session.getAttribute(“name”)； 4、applicationScope：${applicationScope.name}等同与application.getAttribute(“name”)；

通过参数名方式获取：

${page.name};

${request.name};

${session.name};

${application.name}

通过上述的获取时，如果没有指定哪一个域对象的name属性，例如:$(name)，则获取的优先级按照访问域的大小来决定，域小的先被获取。域的范围从小到大一次为，page-request-session-application。

## EL内置对象

EL一共11个内置对象，无需创建即可以使用。这11个内置对象中有10个是Map类型的，最后一个是pageContext对象。

1.pageScope

2requestScope

3.sessionScope

4.applicationScope

5.param；

6.paramValues；

7.header；

8.headerValues；

9.initParam；

10.cookie；

11.pageContext；

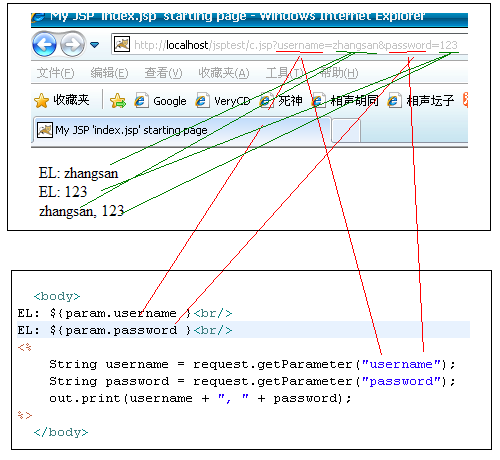
### 域相关内置对象（重点）

注：

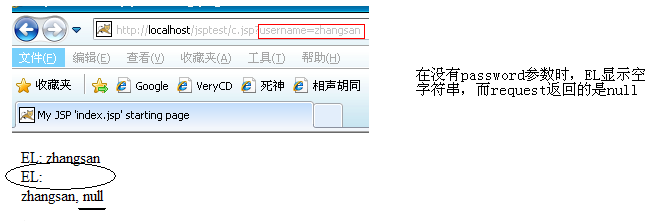
### 请求参数相关内置对象

　　param和paramValues这两个内置对象是用来获取请求参数的。

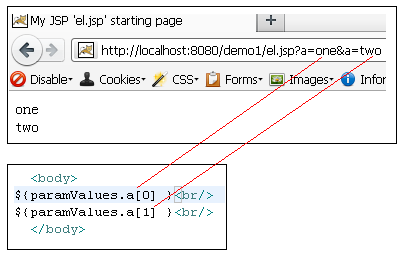
1. param：Map<String,String>类型，param对象可以用来获取参数，与request.getParameter()方法相同。请求参数在底层也是一个map对象，



注意，在使用EL获取参数时，如果参数不存在，返回的是空字符串，而不是null。这一点与使用request.getParameter()方法是不同的。



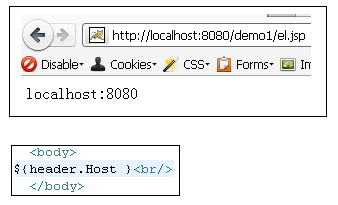
1. paramValues：paramValues是Map<String, String[]>类型，当一个参数名，对应多个参数值时可以使用它。复选框和下拉列表



### 请求头相关内置对象

header和headerValues是与请求头相关的内置对象：

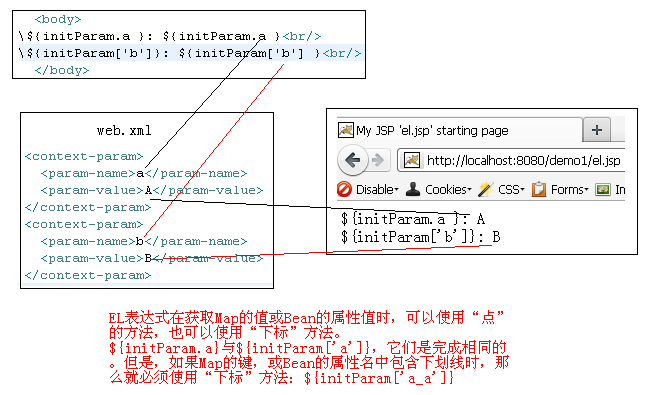
1. header： Map<String,String>类型，用来获取请求头。



1. headerValues：headerValues是Map<String,String[]>类型。当一个请求头名称，对应多个值时，使用该对象，这里就不在赘述。

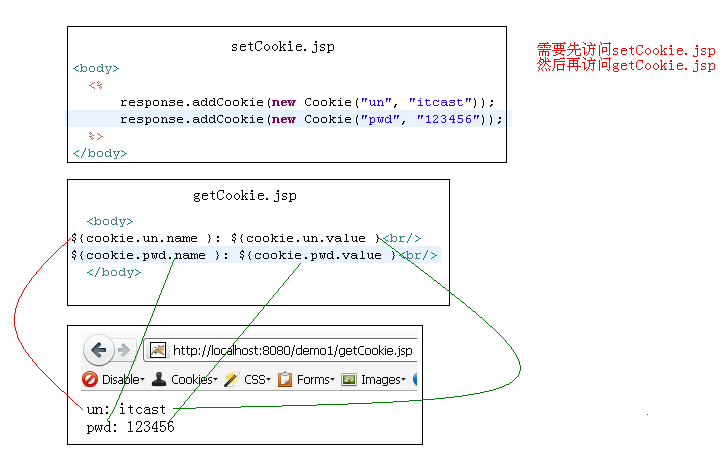
### 应用初始化参数相关内置对象

1. initParam：initParam是Map<String,String>类型。它对应web.xml文件中的<context-param>参数。



### Cookie相关内置对象

* cookie：cookie是Map<String,Cookie>类型，其中key是Cookie的名字，而值是Cookie对象本身。



### pageContext对象

pageContext：pageContext是PageContext类型！可以使用pageContext对象调用getXXX()方法，例如pageContext.getRequest()，可以${pageContext.request}。也就是读取JavaBean属性！！！

作用：通过pageContent获取request请求的相关信息。

|  |  |
| --- | --- |
| **EL表达式** | **说明** |
| ${pageContext.request.queryString} | pageContext.getRequest().getQueryString(); |
| ${pageContext.request.requestURL} | pageContext.getRequest().getRequestURL();  获取资源的请求 |
| ${pageContext.request.contextPath} | pageContext.getRequest().getContextPath();  当前web应用的根路径 |
| ${pageContext.request.method} | pageContext.getRequest().getMethod(); |
| ${pageContext.request.protocol} | pageContext.getRequest().getProtocol(); |
| ${pageContext.request.remoteUser} | pageContext.getRequest().getRemoteUser(); |
| ${pageContext.request.remoteAddr} | pageContext.getRequest().getRemoteAddr(); |
| ${pageContext.session.new} | pageContext.getSession().isNew(); |
| ${pageContext.session.id} | pageContext.getSession().getId(); |
| ${pageContext.servletContext.serverInfo} | pageContext.getServletContext().getServerInfo(); |

## EL函数库

### 1　什么EL函数库

　　EL函数库是由第三方对EL的扩展，我们现在学习的EL函数库是由JSTL添加的。

EL函数库就是定义一些有**返回值**的**静态方法**。然后通过EL语言来调用它们！当然，不只是JSTL可以定义EL函数库，我们也可以自定义EL函数库。

　　EL函数库中包含了很多对字符串的操作方法，以及对集合对象的操作。例如：${fn:length(“abc”)}会输出3，即字符串的长度。

### 2　导入函数库

　　因为是第三方的东西，所以需要导入。导入需要使用taglib指令！

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

### 3　EL函数库介绍

1. String toUpperCase(String input)：
2. String toLowerCase(String input)：
3. int indexOf(String input, String substring)：
4. boolean contains(String input, String substring)：
5. boolean containsIgnoreCase(String input, String substring)：
6. boolean startsWith(String input, String substring)：
7. boolean endsWith(String input, String substring)：
8. String substring(String input, int beginIndex, int endIndex)：
9. String substringAfter(String input, String substring)：hello-world, “-“
10. substringBefore(String input, String substring)：hello-world, “-“
11. String escapeXml(String input)：把字符串的“>”、“<”。。。转义了！
12. String trim(String input)：
13. String replace(String input, String substringBefore, String substringAfter)：
14. String[] split(String input, String delimiters)：
15. int length(Object obj)：可以获取字符串、数组、各种集合的长度！
16. String join(String array[], String separator)：

|  |
| --- |
| <%@taglib prefix=*"fn"* uri=*"http://java.sun.com/jsp/jstl/functions"* %>  …  String[] strs = {"a", "b","c"};  List list = **new** ArrayList();  list.add("a");  pageContext.setAttribute("arr", strs);  pageContext.setAttribute("list", list);  %>  ${fn:length(arr) }<br/><!--3-->  ${fn:length(list) }<br/><!--1-->  ${fn:toLowerCase("Hello") }<br/> <!-- hello -->  ${fn:toUpperCase("Hello") }<br/> <!-- HELLO -->  ${fn:contains("abc", "a")}<br/><!-- true -->  ${fn:containsIgnoreCase("abc", "Ab")}<br/><!-- true -->  ${fn:contains(arr, "a")}<br/><!-- true -->  ${fn:containsIgnoreCase(list, "A")}<br/><!-- true -->  ${fn:endsWith("Hello.java", ".java")}<br/><!-- true -->  ${fn:startsWith("Hello.java", "Hell")}<br/><!-- true -->  ${fn:indexOf("Hello-World", "-")}<br/><!-- 5 -->  ${fn:join(arr, ";")}<br/><!-- a;b;c -->  ${fn:replace("Hello-World", "-", "+")}<br/><!-- Hello+World -->  ${fn:join(fn:split("a;b;c;", ";"), "-")}<br/><!-- a-b-c -->  ${fn:substring("0123456789", 6, 9)}<br/><!-- 678 -->  ${fn:substring("0123456789", 5, -1)}<br/><!-- 56789 -->  ${fn:substringAfter("Hello-World", "-")}<br/><!-- World -->  ${fn:substringBefore("Hello-World", "-")}<br/><!-- Hello -->  ${fn:trim(" a b c ")}<br/><!-- a b c -->  ${fn:escapeXml("<html></html>")}<br/> <!-- <html></html> --> |

## 4　自定义EL函数库

1. 写一个类，写一个有返回值的静态方法；
2. 编写itcast.tld文件，可以参数fn.tld文件来写，把itcast.tld文件放到/WEB-INF目录下；
3. 在页面中添加taglib指令，导入自定义标签库。

ItcastFuncations.java

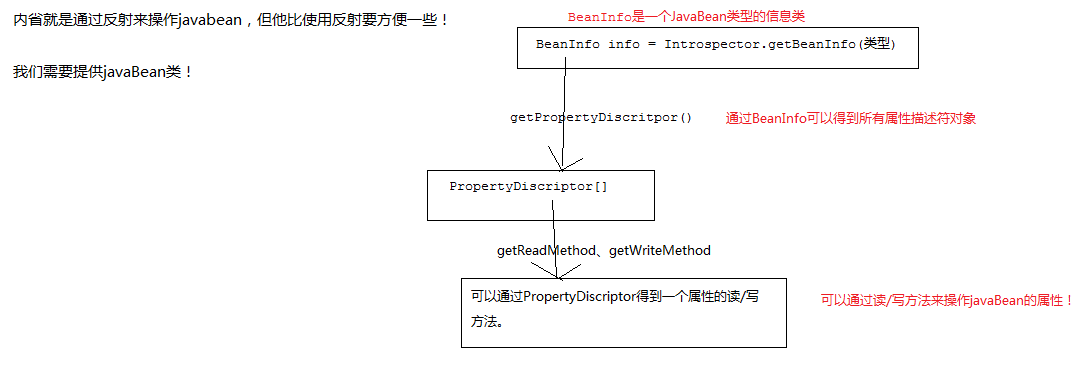
|  |
| --- |
| **package** cn.itcast.el.funcations;  **public** **class** ItcastFuncations {  **public** **static** String test() {  **return** "传智播客自定义EL函数库测试";  }  } |

itcast.tld（放到classes下）

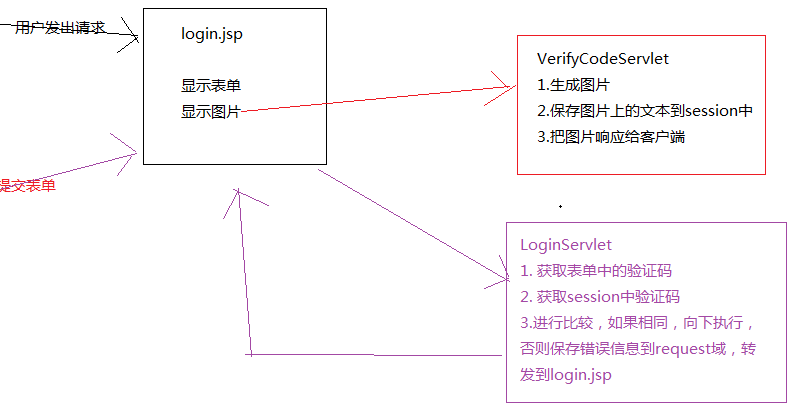
|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <taglib xmlns=*"http://java.sun.com/xml/ns/j2ee"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary\_2\_0.xsd"*  version=*"2.0"*>    <tlib-version>1.0</tlib-version>  <short-name>itcast</short-name>  <uri>http://www.itcast.cn/jsp/functions</uri>  <function>  <name>test</name>  <function-class>cn.itcast.el.funcations.ItcastFuncations</function-class>  <function-signature>String test()</function-signature>  </function>  </taglib> |

index.jsp

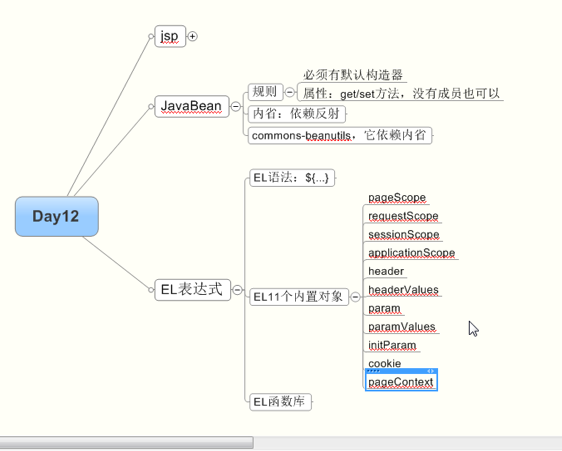
|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <%@ taglib prefix=*"itcast"* uri=*"/WEB-INF/itcast.tld"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <body>  <h1>${itcast:test() }</h1>  </body>  </html> |



内省



验证码思路



# JavaBean

## JavaBean概述

### 什么是JavaBean

JavaBean是一种规范，也就是对类的要求。它要求Java类的成员变量提供getter/setter方法，，这样的成员变量被称之为JavaBean属性。、

属性：属性的名称就是get/set方法去除get和set之后，再把名字小写。

规范要求：

1必须有默认的构造器。

2为成员变量提供getter或者是sertter(两者中提供一个也是可以的)具有getter和setter方法称属性。一般，没有成员变量，但是有get和set方法，也可以称之为属性。

JavaBean还要求类必须提供仅有的无参构造器，例如：public User() {…}

User.java

|  |
| --- |
| **package** cn.itcast.domain;  **public** **class** User {  **private** String username;  **private** String password;  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  } |

### JavaBean属性

JavaBean属性是具有getter/setter方法的成员变量。

1. 也可以只提供getter方法，这样的属性叫只读属性；
2. 也可以只提供setter方法，这样的属性叫只写属性；
3. 如果属性类型为boolean类型，那么读方法的格式可以是get或is。

例如名为abc的boolean类型的属性，它的读方法可以是getAbc()，也可以是isAbc()；

JavaBean属性名要求：前两个字母要么都大写，要么都小写：

|  |
| --- |
| **public** **class** User {  **private** String iD;  **private** String ID;  **private** String qQ;  **private** String QQ;  …  } |

JavaBean可能存在属性，但不存在这个成员变量，例如：

|  |
| --- |
| **public** **class** User {  **public** String getUsername() {  **return** "zhangSan";  }  } |

　　上例中User类有一个名为username的只读属性！但User类并没有username这个成员变量！

还可以并变态一点：

|  |
| --- |
| **public** **class** User {  **private** String hello;    **public** String getUsername() {  **return** hello;  }    **public** **void** setUsername(String username) {  **this**.hello = username;  }  } |

　　上例中User类中有一个名为username的属性，它是可读可写的属性！而Use类的成员变量名为hello！也就是说JavaBean的属性名取决与方法名称，而不是成员变量的名称。但通常没有人做这么变态的事情。

## 内省（了解）

内省的目标是得到JavaBean属性的读、写方法的反射对象，通过反射对JavaBean属性进行操作的一组API。例如User类有名为username的JavaBean属性，通过两个Method对象（一个是getUsenrmae()，一个是setUsername()）来操作User对象。

如果你还不能理解内省是什么，那么我们通过一个问题来了解内省的作用。现在我们有一个Map，内容如下：

|  |
| --- |
| Map<String,String> map = **new** HashMap<String,String>();  map.put("username", "admin");  map.put("password", "admin123"); |
| **public** **class** User {  **private** String username;  **private** String password;  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** User() {  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

现在需要把map的数据封装到一个User对象中！User类有两个JavaBean属性，一个叫username，另一个叫password。

你可能想到的是反射，通过map的key来查找User类的Field！这么做是没有问题的，但我们要知道类的成员变量是私有的，虽然也可以通过反射去访问类的私有的成员变量，但我们也要清楚反射访问私有的东西是有“危险”的，所以还是建议通过getUsername和setUsername来访问JavaBean属性。

### 内省之获取BeanInfo

我们这里不想去对JavaBean规范做过多的介绍，所以也就不在多介绍BeanInfo的“出身”了。你只需要知道如何得到它，以及BeanInfo有什么。

通过java.beans.Introspector的getBeanInfo()方法来获取java.beans.BeanInfo实例。

|  |
| --- |
| BeanInfo beanInfo = Introspector.*getBeanInfo*(User.**class**); |

### 得到所有属性描述符（PropertyDescriptor）

通过BeanInfo可以得到这个类的所有JavaBean属性的PropertyDescriptor对象。然后就可以通过PropertyDescriptor对象得到这个属性的getter/setter方法的Method对象了。

|  |
| --- |
| PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors(); |

每个PropertyDescriptor对象对应一个JavaBean属性：

1. String getName()：获取JavaBean属性名称；
2. Method getReadMethod：获取属性的读方法；
3. Method getWriteMethod：获取属性的写方法。

### 完成Map数据封装到User对象中

|  |
| --- |
| **public** **void** fun1() **throws** Exception {  Map<String,String> map = **new** HashMap<String,String>();  map.put("username", "admin");  map.put("password", "admin123");    BeanInfo beanInfo = Introspector.*getBeanInfo*(User.**class**);    PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();    User user = **new** User();  **for**(PropertyDescriptor pd : pds) {  String name = pd.getName();  String value = map.get(name);  **if**(value != **null**) {  Method writeMethod = pd.getWriteMethod();  writeMethod.invoke(user, value);  }  }    System.*out*.println(user);  } |

## 3commons-beanutils

提到内省，不能不提commons-beanutils这个工具。它底层使用了内省，对内省进行了大量的简化！

使用beanutils需要的jar包：

1. commons-beanutils.jar；
2. commons-logging.jar；

### 设置JavaBean属性

|  |
| --- |
| User user = **new** User();    BeanUtils.*setProperty*(user, "username", "admin");  BeanUtils.*setProperty*(user, "password", "admin123");    System.*out*.println(user); |

### 获取JavaBean属性

|  |
| --- |
| User user = **new** User("admin", "admin123");    String username = BeanUtils.*getProperty*(user, "username");  String password = BeanUtils.*getProperty*(user, "password");    System.*out*.println("username=" + username + ", password=" + password); |

### 封装Map数据到JavaBean对象中

|  |
| --- |
| Map<String,String> map = **new** HashMap<String,String>();  map.put("username", "admin");  map.put("password", "admin123");    User user = **new** User();  BeanUtils.*populate*(user, map);    System.*out*.println(user); |

## JSP与JavaBean相关的动作标签

在JSP中与JavaBean相关的标签有：

<jsp:useBean>：创建JavaBean对象；

<jsp:setProperty>：设置JavaBean属性；

<jsp:getProperty>：获取JavaBean属性；

我们需要先创建一个JavaBean类：

User.java

|  |
| --- |
| **package** cn.itcast.domain;  **public** **class** User {  **private** String username;  **private** String password;  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** User() {  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

### <jsp:useBean>

<jsp:useBean>标签的作用是创建JavaBean对象：

1. 在当前JSP页面创建JavaBean对象；
2. 把创建的JavaBean对象保存到域对象中；

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* /> |

上面代码表示在当前JSP页面中创建User类型的对象，并且把它保存到page域中了。下面我们把<jsp:useBean>标签翻译成Java代码：

|  |
| --- |
| <%  cn.itcast.domain.User user1 = **new** cn.itcast.domain.User();  pageContext.setAttribute("user1", user1);  %> |

这说明我们可以在JSP页面中完成下面的操作：

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* />  <%=user1 %>  <%  out.println(pageContext.getAttribute("user1"));  %> |

<jsp:useBean>标签默认是把JavaBean对象保存到page域，还可以通过scope标签属性来指定保存的范围：

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* scope=*"page"*/>  <jsp:useBean id=*"user2"* class=*"cn.itcast.domain.User"* scope=*"request"*/>  <jsp:useBean id=*"user3"* class=*"cn.itcast.domain.User"* scope=*"session"*/>  <jsp:useBean id=*"user4"* class=*"cn.itcast.domain.User"* scope=*"applicatioin"*/> |

<jsp:useBean>标签其实不一定会创建对象！！！其实它会先在指定范围中查找这个对象，如果对象不存在才会创建，我们需要重新对它进行翻译：

|  |
| --- |
| <jsp:useBean id=*"user4"* class=*"cn.itcast.domain.User"* scope=*"applicatioin"*/> |
| <%  cn.itcast.domain.User user4 = (cn.itcast.domain.User)application.getAttribute("user4");  **if**(user4 == **null**) {  user4 = **new** cn.itcast.domain.User();  application.setAttribute("user4", user4);  }  %> |

### <jsp:setProperty>和<jsp:getProperty>

<jsp:setProperty>标签的作用是给JavaBean设置属性值，而<jsp:getProperty>是用来获取属性值。在使用它们之前需要先创建JavaBean：

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* />  <jsp:setProperty property=*"username"* name=*"user1"* value=*"admin"*/>  <jsp:setProperty property=*"password"* name=*"user1"* value=*"admin123"*/>  用户名：<jsp:getProperty property=*"username"* name=*"user1"*/><br/>  密　码：<jsp:getProperty property=*"password"* name=*"user1"*/><br/> |

# JSTL标签库

## JSTL标签库

### 什么是JSTL

JSTL是apache对EL表达式的扩展（也就是说JSTL依赖EL），JSTL是标签语言！JSTL标签使用以来非常方便，它与JSP动作标签一定，只不过它不是JSP内置的标签，需要我们自己导包，以及指定标签库而已！

如果你使用MyEclipse开发JavaWeb，那么在把项目发布到Tomcat时，你会发现，MyEclipse会在lib目录下存放jstl的Jar包！如果你没有使用MyEclipse开发那么需要自己来导入这个JSTL的Jar包：jstl-1.2.jar。

### JSTL标签库

JSTL一共包含四大标签库：

core：核心标签库，我们学习的重点；

fmt：格式化标签库，只需要学习两个标签即可；

sql：数据库标签库，不需要学习了，它过时了；

xml：xml标签库，不需要学习了，它过时了。

## 使用taglib指令导入标签库

除了JSP动作标签外，使用其他第三方的标签库都需要：

1、导包；

2、在使用标签的JSP页面中使用taglib指令导入标签库；

下面是导入JSTL的core标签库：

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

prefix="c"：指定标签库的前缀，这个前缀可以随便给值，但大家都会在使用core标签库时指定前缀为c；

uri="http://java.sun.com/jstl/core"：指定标签库的uri，它不一定是真实存在的网址，但它可以让JSP找到标签库的描述文件；

## core标签库常用标签

### out和set

　out

|  |  |
| --- | --- |
| <c:out value=”aaa”/> | 输出aaa字符串常量 |
| <c:out value=”${aaa}”/> | 与${aaa}相同 |
| <c:out value=”${aaa}” default=”xxx”/> | 当${aaa}不存在时，输出xxx字符串 |
| <%  request.setAttribute("a","<script>alert('hello');</script>");  %>  <c:out value="${a }" default="xxx" escapeXml="false" /> | 当escapeXml为false，不会转换“<”、“>”。这可能会受到JavaScript攻击。 |

set：<c:set var=”属性” value=”属性值”/>

|  |  |
| --- | --- |
| <c:set var=”a” value=”hello”/> | 在pageContext中添加name为a，value为hello的数据。 |
| <c:set var=”a” value=”hello” scope=”session”/> | 在session中添加name为a，value为hello的数据。如果不写scope，则默认为pageContext添加属性 |

### remove

|  |  |
| --- | --- |
| <%  pageContext.setAttribute("a", "pageContext");  request.setAttribute("a", "session");  session.setAttribute("a", "session");  application.setAttribute("a", "application");  %>  <c:remove var="a"/>  <c:out value="${a }" default="none"/> | 删除所有域中name为a的数据！ |
| <c:remove var="a" scope=”page”/> | 删除pageContext中name为a的数据！ |

### url标签

　　url标签会在需要URL重写时添加sessionId。

|  |  |
| --- | --- |
| <c:url value="/"/> | 输出上下文路径：/day08\_01/ |
| <c:url value="/" var="a" scope="request"/> | 把本该输出的结果赋给变量a。范围为request |
| <c:url value="/AServlet"/> | 输出：/day08\_01/AServlet |
| <c:url value="/AServlet">  <c:param name="username" value="abc"/>  <c:param name="password" value="123"/>  </c:url> | 输出：/day08\_01/AServlet?username=abc&password=123  如果参数中包含中文，那么会自动使用URL编码！ |

### if 标签

if标签的test属性必须是一个boolean类型的值，如果test的值为true，那么执行if标签的内容，否则不执行。

|  |
| --- |
| <c:set var=*"a"* value=*"hello"* />  <c:if test="${**not empty** a }">  <c:out value="${a }"/>//输出内容  </c:if> |

### choose标签

choose标签对应Java中的if/else if/else结构。when标签的test为true时，会执行这个when的内容。当所有when标签的test都为false时，才会执行otherwise标签的内容。

|  |
| --- |
| <c:set var=*"score"* value="${param.score }"/>  <c:choose>  <c:when test="${score > 100 || score < 0}">错误的分数：${score }</c:when>  <c:when test="${score >= 90 }">A级</c:when>  <c:when test="${score >= 80 }">B级</c:when>  <c:when test="${score >= 70 }">C级</c:when>  <c:when test="${score >= 60 }">D级</c:when>  <c:otherwise>E级</c:otherwise>  </c:choose> |

### forEach

forEach当前就是循环标签了，forEach标签有多种两种使用方式：

使用循环变量，指定开始和结束值，类似for(int i = 1; i <= 10; i++) {}；

循环遍历集合，类似for(Object o : 集合)；

属性：

\* var：循环变量

\* begin：设置循环变量从几开始。

\* end：设置循环变量到几结束。

\* step：设置步长！等同与java中的i++，或i+=2。step默认为1

循环变量方式：

|  |
| --- |
| <c:set var=*"sum"* value=*"0"* />  <c:forEach var=*"i"* begin=*"1"* end=*"10"*>  <c:set var=*"sum"* value="${sum + i}" />  </c:forEach>  <c:out value="*sum =* ${sum }"/> |
| <c:set var=*"sum"* value=*"0"* />  <c:forEach var=*"i"* begin=*"1"* end=*"10"* step=*"2"*>  <c:set var=*"sum"* value="${sum + i}" />  </c:forEach>  <c:out value="*sum =* ${sum }"/> |

遍历集合或数组方式：

|  |
| --- |
| <%  String[] names = {"zhangSan", "liSi", "wangWu", "zhaoLiu"};  pageContext.setAttribute("ns", names);  %>  <c:forEach var=*"item"* items="${ns}">  <c:out value="*name:* ${item }"/><br/>  </c:forEach> |

遍历List

|  |
| --- |
| <%  List<String> names = **new** ArrayList<String>();  names.add("zhangSan");  names.add("liSi");  names.add("wangWu");  names.add("zhaoLiu");  pageContext.setAttribute("ns", names);  %>  <c:forEach var=*"item"* items="${ns }">  <c:out value="*name:* ${item }"/><br/>  </c:forEach>  ${list.size}//这里可以调用list中的方法。 |

遍历Map

|  |
| --- |
| <%  Map<String,String> stu = **new** LinkedHashMap<String,String>();  stu.put("number", "N\_1001");  stu.put("name", "zhangSan");  stu.put("age", "23");  stu.put("sex", "male");  pageContext.setAttribute("stu", stu);  %>  <c:forEach var=*"item"* items="${stu }">  <c:out value="${item.key }*:* ${item.value }"/><br/>  </c:forEach> |

forEach标签还有一个属性：varStatus，这个属性用来指定接收“循环状态”的对象名，例如：<forEach varStatus=”vs” …/>，这时就可以使用vs这个对象来获取循环的状态了。 count：int类型，循环的次数，当前以遍历元素的个数；

index：int类型，循环的索引，从0开始，当前元素的下标；

first：boolean类型，是否为第一个元素；

last：boolean类型，是否为最后一个元素；

current：Object类型，表示当前项目。

begin：遍历的开始位置是(索引)，从0开始。

end：遍历的结束位置，如果省略了end元素，表示一直遍历到结尾的元素。

step：循环的步长，每次增加或减小的索引。

|  |
| --- |
| <c:forEach var=*"item"* items="${ns }" varStatus=*"vs"*>  <c:if test="${vs.first }">第一行：</c:if>  <c:if test="${vs.last }">最后一行：</c:if>  <c:out value="*第*${vs.count }*行:* "/>  <c:out value="*[*${vs.index }*]:* "/>  <c:out value="*name:* ${vs.current }"/><br/>  </c:forEach> |

for-each标签输出一组连续的数字

var：指定每个数字的名称，jsp引擎将他作为key把数字存到page域中。

vegin：要输出的初始值;

end：结束的值；

step：可选的步长；

|  |
| --- |
|  |

### fmt标签库常用标签

fmt标签库是用来格式化输出的，通常需要格式化的有时间和数字。

格式化时间：

|  |
| --- |
| <%@ taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>  ......  <%  Date date = **new** Date();  pageContext.setAttribute("d", date);  %>  <fmt:formatDate value="${d }" pattern=*"yyyy-MM-dd HH:mm:ss"*/> |

格式化数字：

|  |
| --- |
| <%  **double** d1 = 3.5;  **double** d2 = 4.4;  pageContext.setAttribute("d1", d1);  pageContext.setAttribute("d2", d2);  %>  <fmt:formatNumber value="${d1 }" pattern=*"0.00"*/><br/>  <fmt:formatNumber value="${d2 }" pattern=*"#.##"*/> |

### forTokens

## 自定义标签

###### 1　自定义标签概述

1.1　自定义标签的步骤

其实我们在JSP页面中使用标签就等于调用某个对象的某个方法一样，例如：<c:if test=””>，这就是在调用对象的方法一样。自定义标签其实就是自定义类一样！

* 定义标签处理类：必须是Tag或SimpleTag的实现类；
* 编写标签库描述符文件（TLD）；

SimpleTag接口是JSP2.0中新给出的接口，用来简化自定义标签，所以现在我们基本上都是使用SimpleTag。

Tag是老的，传统的自定义标签时使用的接口，现在不建议使用它了。

1.2　SimpleTag接口介绍

SimpleTag接口内容如下：

1. void doTag()：标签执行方法；
2. JspTag getParent()：获取父标签；
3. void setParent(JspTag parent)：设置父标签
4. void setJspContext(JspContext context)：设置PageContext
5. void setJspBody(JspFragment jspBody)：设置标签体对象；

请记住，万物皆对象！在JSP页面中的标签也是对象！你可以通过查看JSP的“真身”清楚的知道，所有标签都会变成对象的方法调用。标签对应的类我们称之为“标签处理类”！

标签的生命周期：

* 当容器（Tomcat）第一次执行到某个标签时，会创建标签处理类的实例；
* 然后调用setJspContext(JspContext)方法，把当前JSP页面的pageContext对象传递给这个方法；
* 如果当前标签有父标签，那么使用父标签的标签处理类对象调用setParent(JspTag)方法；
* 如果标签有标签体，那么把标签体转换成JspFragment对象，然后调用setJspBody()方法；
* 每次执行标签时，都调用doTag()方法，它是标签处理方法。

HelloTag.java

|  |
| --- |
| **public** **class** HelloTag **implements** SimpleTag {  **private** JspTag parent;  **private** PageContext pageContext;  **private** JspFragment jspBody;    **public** **void** doTag() **throws** JspException, IOException {  pageContext.getOut().print("Hello Tag!!!");  }  **public** **void** setParent(JspTag parent) {  **this**.parent = parent;  }  **public** JspTag getParent() {  **return** **this**.parent;  }  **public** **void** setJspContext(JspContext pc) {  **this**.pageContext = (PageContext) pc;  }  **public** **void** setJspBody(JspFragment jspBody) {  **this**.jspBody = jspBody;  }  } |

1.3　标签库描述文件（TLD）

标签库描述文件是用来描述当前标签库中的标签的！标签库描述文件的扩展名为tld，你可以把它放到WEB-INF下，这样就不会被客户端直接访问到了。

hello.tld

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <taglib version=*"2.0"* xmlns=*"http://java.sun.com/xml/ns/j2ee"*  xmlns:xml=*"http://www.w3.org/XML/1998/namespace"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/j2ee*  *http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary\_2\_0.xsd "*>  <tlib-version>1.0</tlib-version>  <short-name>itcast</short-name>  <uri>http://www.itcast.cn/tags</uri>  <tag>  <name>hello</name>  <tag-class>cn.itcast.tag.HelloTag</tag-class>  <body-content>empty</body-content>  </tag>  </taglib> |

1.4　使用标签

在页面中使用标签分为两步：

1. 使用taglib导入标签库；
2. 使用标签；

|  |
| --- |
| <%@ taglib prefix=*"it"* uri=*"/WEB-INF/hello.tld"* %>  ......  <it:hello/> |

###### 2　自定义标签进阶

2.1　继承SimpleTagSupport

　　继承SimpleTagSuppport要比实现SimpleTag接口方便太多了，现在你只需要重写doTag()方法，其他方法都已经被SimpleTagSuppport完成了。

|  |
| --- |
| **public** **class** HelloTag **extends** SimpleTagSupport {  **public** **void** doTag() **throws** JspException, IOException {  **this**.getJspContext().getOut().write("<p>Hello SimpleTag!</p>");  }  } |

2.2　有标签体的标签

我们先来看看标签体内容的可选值：

<body-content>元素的可选值有：

1. empty：无标签体。
2. JSP：传统标签支持它，**SimpleTag已经不再支持使用<body-content>JSP</body-content>。**标签体内容可以是任何东西：EL、JSTL、<%=%>、<%%>，以及html；
3. scriptless：标签体内容不能是Java脚本，但可以是EL、JSTL等。在SimpleTag中，**如果需要有标签体，那么就使用该选项**；
4. tagdependent：标签体内容不做运算，由标签处理类自行处理，无论标签体内容是EL、JSP、JSTL，都不会做运算。**这个选项几乎没有人会使用！**

自定义有标签体的标签需要：

1. 获取标签体对象：JspFragment jspBody = getJspBody();；
2. 把标签体内容输出到页面：jspBody.invoke(null)；
3. tld中指定标签内容类型：scriptless。

|  |
| --- |
| **public** **class** HelloTag **extends** SimpleTagSupport {  **public** **void** doTag() **throws** JspException, IOException {  PageContext pc = (PageContext) **this**.getJspContext();  HttpServletRequest req = (HttpServletRequest) pc.getRequest();  String s = req.getParameter("exec");  **if**(s != **null** && s.endsWith("true")) {  JspFragment body = **this**.getJspBody();  body.invoke(null);  }  }  } |
| <tag>  <name>hello</name>  <tag-class>cn.itcast.tags.HelloTag</tag-class>  <body-content>scriptless</body-content>  </tag> |
| <itcast:hello>  <h1>哈哈哈~</h1>  </itcast:hello> |

2.3　不执行标签下面的页面内容

　　如果希望在执行了自定义标签后，不再执行JSP页面下面的东西，那么就需要在doTag()方法中使用SkipPageException。

|  |
| --- |
| **public** **class** SkipTag **extends** SimpleTagSupport {  **public** **void** doTag() **throws** JspException, IOException {  **this**.getJspContext().getOut().print("<h1>只能看到我！</h1>");  **throw** **new** SkipPageException();  }  } |
| <tag>  <name>skip</name>  <tag-class>cn.itcast.tags.SkipTag</tag-class>  <body-content>empty</body-content>  </tag> |
| <itcast:skip/>  <h1>看不见我！</h1> |

2.4　带有属性的标签

　　一般标签都会带有属性，例如<c:if test=””>，其中test就是一个boolean类型的属性。完成带有属性的标签需要：

1. 在处理类中给出JavaBean属性（提供get/set方法）；
2. 在TLD中部属相关属性。

|  |
| --- |
| **public** **class** IfTag **extends** SimpleTagSupport {  **private** **boolean** test;  **public** **boolean** isTest() {  **return** test;  }  **public** **void** setTest(**boolean** test) {  **this**.test = test;  }  @Override  **public** **void** doTag() **throws** JspException, IOException {  **if**(test) {  **this**.getJspBody().invoke(**null**);  }  }  } |
| <tag>  <name>if</name>  <tag-class>cn.itcast.tag.IfTag</tag-class>  <body-content>scriptless</body-content>  <attribute>  <name>test</name>  <required>true</required>  <rtexprvalue>true</rtexprvalue>  </attribute>  </tag> |
| <%  pageContext.setAttribute("one", **true**);  pageContext.setAttribute("two", **false**);  %>  <it:if test="${one }">xixi</it:if>  <it:if test="${two }">haha</it:if>  <it:if test=*"true"*>hehe</it:if> |

# 监听器

## 监听器

监听就是监视行为。在Web系统中，所谓的监听器就是应用监听事件来监听请求中的行为而创建的一组类。HttpServletRequest、HttpSession、ServletContext对象在Web容器中遵循生成、运行、销毁这样的生命周期。当进行相关的监听配置后，Web容器就会调用监听器上的方法，进行对应的事件处理，从而了解运行的情况或者运行其他的程序。各监听器接口和事件类如表7.5所示。



使用监听器需要实现相应的监听接口。在触发监听事件时，应用服务器会自动调用监

听方法。开发人员不需要关心应用服务器如何调用，只需要实现这些方法就行。

## JavaWeb监听器概述

在JavaWeb被监听的事件源为：ServletContext、HttpSession、ServletRequest，即三大域对象。

 监听域对象“创建”与“销毁”的监听器；

 监听域对象“操作域属性”的监听器；(setAttribute或getAttribute)

 监听HttpSession的监听器。

## 创建与销毁监听器

### ServletContextListener：继承ServletContextListener接口，重写下面的两个方法，当Tomcat启动和关闭时调用下面两个方法，执行重写后方法中的代码

ServletContext对象被创建后调用：

 public void contextInitialized(ServletContextEvent evt)：

ServletContext对象被销毁前调用；

 public void contextDestroyed(ServletContextEvent evt)：

### HttpSessionListener：开始会话和结束会话时调用下面两个方法。

public void sessionCreated(HttpSessionEvent evt)：HttpSession对象被创建后调用；

public void sessionDestroyed(HttpSessionEvent evt)：HttpSession对象被销毁前调用；

## JavaWeb监听器

三大组件：

* Servlet
* Listener
* Filter

Listener：监听器

1. 初次相见：AWT
2. 二次相见：SAX

监听器：

* 它是一个接口，内容由我们来实现；
* 它需要注册，例如注册在按钮上！
* 监听器中的方法，会在特殊事件发生时被调用！

观察者：

* 事件源；
* 小偷
* 事件；
* 偷东西
* 监听器；
* 警察
* 监听器中的方法：抓捕

JavaWeb中的监听器

* 事件源：三大域！

### ServletContext

生命周期监听：ServletContextListener，它有两个方法，一个在出生时调用，一个在死亡时调用；

void **contextInitialized**([ServletContextEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletContextEvent.html) sce)：创建SErvletcontext时

void **contextDestroyed**([ServletContextEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletContextEvent.html) sce)：销毁Servletcontext时

属性监听：ServletContextAttributeListener，它有三个方法，一个在添加属性时调用，一个在替换属性时调用，最后一个是在移除属性时调用。

void **attributeAdded**([ServletContextAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletContextAttributeEvent.html) event)：添加属性时；

void **attributeReplaced**([ServletContextAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletContextAttributeEvent.html) event)：替换属性时；

void **attributeRemoved**([ServletContextAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletContextAttributeEvent.html) event)：移除属性时；

### HttpSession

生命周期监听：HttpSessionListener，它有两个方法，一个在出生时调用，一个在死亡时调用；第一次调用request.getsession();或者第一次访问jsp页面时会调用。

void **sessionCreated**([HttpSessionEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/http/HttpSessionEvent.html) se)：创建session时

void **sessionDestroyed**([HttpSessionEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/http/HttpSessionEvent.html) se)：销毁session时

属性监听：HttpSessioniAttributeListener，它有三个方法，一个在添加属性时调用，一个在替换属性时调用，最后一个是在移除属性时调用。

void **attributeAdded**([HttpSessionBindingEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/http/HttpSessionBindingEvent.html) event)：添加属性时；

void **attributeReplaced**([HttpSessionBindingEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/http/HttpSessionBindingEvent.html) event)：替换属性时

void **attributeRemoved**([HttpSessionBindingEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/http/HttpSessionBindingEvent.html) event)：移除属性时

### ServletRequest

生命周期监听：ServletRequestListener，它有两个方法，一个在出生时调用，一个在死亡时调用；

void **requestInitialized**([ServletRequestEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletRequestEvent.html) sre)：创建request时

void **requestDestroyed**([ServletRequestEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletRequestEvent.html) sre)：销毁request时

属性监听：ServletRequestAttributeListener，它有三个方法，一个在添加属性时调用，一个在替换属性时调用，最后一个是在移除属性时调用。

void **attributeAdded**([ServletRequestAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletRequestAttributeEvent.html) srae)：添加属性时

void **attributeReplaced**([ServletRequestAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletRequestAttributeEvent.html) srae)：替换属性时

void **attributeRemoved**([ServletRequestAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletRequestAttributeEvent.html) srae)：移除属性时

### javaWeb中完成 编写监听器：

* 写一个监听器类：要求必须去实现某个监听器接口；
* 注册，是在web.xml中配置来完成注册！

### 事件对象：

* ServletContextEvent：ServletContext getServletContext()
* HttpSessionEvent：HttpSession getSession()
* ServletRequest：
* ServletContext getServletContext()；
* ServletReques getServletRequest()；
* ServletContextAttributeEvent：
* ServletContext getServletContext()；
* String getName()：获取属性名
* Object getValue()：获取属性值
* [HttpSessionBindingEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/http/HttpSessionBindingEvent.html)：略
* [ServletRequestAttributeEvent](mk:@MSITStore:F:\F\帮助\j2EE6.0.chm::/javax/servlet/ServletRequestAttributeEvent.html) ：略

感知监听（都与HttpSession相关）

* 它用来添加到JavaBean上，而不是添加到三大域上！
* 这两个监听器都不需要在web.xml中注册！

HttpSessionBindingListener：添加到javabean上，javabean就知道自己是否添加到session中了。

## 事件对象

* ServletContextEvent：ServletContext getServletContext()；
* HttpSeessionEvent：HttpSession getSession()；
* ServletRequestEvent：
* ServletRequest getServletRequest()
* ServletContext getServletContext()

编写测试例子：

* 编写MyServletContextListener类，实现ServletContextListener接口；
* 在web.xml文件中部署监听器；
* 为了看到session销毁的效果，在web.xml文件中设置session失效时间为1分钟；

|  |
| --- |
| /\*  \* ServletContextListener实现类  \* contextDestroyed() -- 在ServletContext对象被销毁前调用  \* contextInitialized()--在ServletContext对象被创建后调用  \* ServletContextEvent -- 事件类对象  \* 该类有getServletContext()，用来获取ServletContext对象，即获取事件源对象  \*/  **public** **class** MyServletContextListener **implements** ServletContextListener {  **public** **void** contextDestroyed(ServletContextEvent evt) {  System.*out*.println("销毁ServletContext对象");  }  **public** **void** contextInitialized(ServletContextEvent evt) {  System.*out*.println("创建ServletContext对象");  }  } |
| /\*  \* HttpSessionListener实现类  \* sessionCreated() -- 在HttpSession对象被创建后被调用  \* sessionDestroyed() -- -- 在HttpSession对象被销毁前调用  \* HttpSessionEvent -- 事件类对象  \* 该类有getSession()，用来获取当前HttpSession对象，即获取事件源对象  \*/  **public** **class** MyHttpSessionListener **implements** HttpSessionListener {  **public** **void** sessionCreated(HttpSessionEvent evt) {  System.*out*.println("创建session对象");  }  **public** **void** sessionDestroyed(HttpSessionEvent evt) {  System.*out*.println("销毁session对象");  }  } |
| /\*  \* ServletRequestListener实现类  \* requestDestroyed() -- 在ServletRequest对象被销毁前调用  \* requestInitialized() -- 在ServletRequest对象被创建后调用  \* ServletRequestEvent -- 事件类对象  \* 该类有getServletContext()，用来获取ServletContext对象  \* 该类有getServletRequest()，用来获取当前ServletRequest对象，即事件源对象  \*/  **public** **class** MyServletRequestListener **implements** ServletRequestListener {  **public** **void** requestDestroyed(ServletRequestEvent evt) {  System.*out*.println("销毁request对象");  }  **public** **void** requestInitialized(ServletRequestEvent evt) {  System.*out*.println("创建request对象");  }  } |
| <listener>  <listener-class>cn.itcast.listener.MyServletContextListener</listener-class>  </listener>  <listener>  <listener-class>cn.itcast.listener.MyHttpSessionListener</listener-class>  </listener>  <listener>  <listener-class>cn.itcast.listener.MyServletRequestListener</listener-class>  </listener>  <session-config>  <session-timeout>1</session-timeout>  </session-config> |

## 操作域属性的监听器

当对域属性进行增、删、改时，执行的监听器一共有三个：

* ServletContextAttributeListener：在ServletContext域进行增、删、改属性时调用下面方法。
* public void attributeAdded(ServletContextAttributeEvent evt)
* public void attributeRemoved(ServletContextAttributeEvent evt)
* public void attributeReplaced(ServletContextAttributeEvent evt)
* HttpSessionAttributeListener：在HttpSession域进行增、删、改属性时调用下面方法
* public void attributeAdded(HttpSessionBindingEvent evt)
* public void attributeRemoved (HttpSessionBindingEvent evt)
* public void attributeReplaced (HttpSessionBindingEvent evt)
* ServletRequestAttributeListener：在ServletRequest域进行增、删、改属性时调用下面方法
* public void attributeAdded(ServletRequestAttributeEvent evt)
* public void attributeRemoved (ServletRequestAttributeEvent evt)
* public void attributeReplaced (ServletRequestAttributeEvent evt)

下面对这三个监听器的事件对象功能进行介绍：

* ServletContextAttributeEvent
* String getName()：获取当前操作的属性名；
* Object getValue()：获取当前操作的属性值；
* ServletContext getServletContext()：获取ServletContext对象。
* HttpSessionBindingEvent
* String getName()：获取当前操作的属性名；
* Object getValue()：获取当前操作的属性值；
* HttpSession getSession()：获取当前操作的session对象。
* ServletRequestAttributeEvent
* String getName()：获取当前操作的属性名；
* Object getValue()：获取当前操作的属性值；
* ServletContext getServletContext()：获取ServletContext对象；
* ServletRequest getServletRequest()：获取当前操作的ServletRequest对象。

|  |
| --- |
| **public** **class** MyListener **implements** ServletContextAttributeListener,  ServletRequestAttributeListener, HttpSessionAttributeListener {  **public** **void** attributeAdded(HttpSessionBindingEvent evt) {  System.*out*.println("向session中添加属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeRemoved(HttpSessionBindingEvent evt) {  System.*out*.println("从session中移除属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeReplaced(HttpSessionBindingEvent evt) {  System.*out*.println("修改session中的属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeAdded(ServletRequestAttributeEvent evt) {  System.*out*.println("向request中添加属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeRemoved(ServletRequestAttributeEvent evt) {  System.*out*.println("从request中移除属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeReplaced(ServletRequestAttributeEvent evt) {  System.*out*.println("修改request中的属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeAdded(ServletContextAttributeEvent evt) {  System.*out*.println("向context中添加属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeRemoved(ServletContextAttributeEvent evt) {  System.*out*.println("从context中移除属性：" + evt.getName() + "=" + evt.getValue());  }  **public** **void** attributeReplaced(ServletContextAttributeEvent evt) {  System.*out*.println("修改context中的属性：" + evt.getName() + "=" + evt.getValue());  }  } |
| **public** **class** ListenerServlet **extends** BaseServlet {  **public** String contextOperation(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  ServletContext context = **this**.getServletContext();  context.setAttribute("a", "a");  context.setAttribute("a", "A");  context.removeAttribute("a");  **return** "/index.jsp";  }    ///////////////////////////////    **public** String sessionOperation(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  HttpSession session = request.getSession();  session.setAttribute("a", "a");  session.setAttribute("a", "A");  session.removeAttribute("a");  **return** "/index.jsp";  }  ///////////////////////////////    **public** String requestOperation(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setAttribute("a", "a");  request.setAttribute("a", "A");  request.removeAttribute("a");  **return** "/index.jsp";  }  } |
| <body>  <a href=*"*<c:url value=*'/ListenerServlet?method=contextOperation'*/>*"*>SevletContext操作属性</a>  <br/>  <a href=*"*<c:url value=*'/ListenerServlet?method=sessionOperation'*/>*"*>HttpSession操作属性</a>  <br/>  <a href=*"*<c:url value=*'/ListenerServlet?method=requestOperation'*/>*"*>ServletRequest操作属性</a> |  </body> |

## HttpSession的监听器

还有两个与HttpSession相关的特殊的监听器，这两个监听器的特点如下：

* 不用在web.xml文件中部署；
* 这两个监听器不是给session添加，而是给Bean添加。即让Bean类实现监听器接口，然后再把Bean对象添加到session域中。

下面对这两个监听器介绍一下：

* HttpSessionBindingListener：当某个类实现了该接口后，可以感知本类对象添加到session中，以及感知从session中移除。例如让Person类实现HttpSessionBindingListener接口，那么当把Person对象添加到session中，或者把Person对象从session中移除时会调用下面两个方法：
* public void valueBound([HttpSessionBindingEvent](mk:@MSITStore:F:\帮助\J2EE1.5.chm::/javax/servlet/http/HttpSessionBindingEvent.html) event)：当把监听器对象添加到session中会调用监听器对象的本方法；
* public void valueUnbound([HttpSessionBindingEvent](mk:@MSITStore:F:\帮助\J2EE1.5.chm::/javax/servlet/http/HttpSessionBindingEvent.html) event)：当把监听器对象从session中移除时会调用监听器对象的本方法；

这里要注意，HttpSessionBindingListener监听器的使用与前面介绍的都不相同，当该监听器对象添加到session中，或把该监听器对象从session移除时会调用监听器中的方法。并且无需在web.xml文件中部署这个监听器。

示例步骤：

* 编写Person类，让其实现HttpSessionBindingListener监听器接口；
* 编写Servlet类，一个方法向session中添加Person对象，另一个从session中移除Person对象；
* 在index.jsp中给出两个超链接，分别访问Servlet中的两个方法。

Pseron.java

|  |
| --- |
| **public** **class** Person **implements** HttpSessionBindingListener {  **private** String name;  **private** **int** age;  **private** String sex;    **public** Person(String name, **int** age, String sex) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.sex = sex;  }  **public** Person() {  **super**();  }  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** String getSex() {  **return** sex;  }  **public** **void** setSex(String sex) {  **this**.sex = sex;  }  **public** **void** valueBound(HttpSessionBindingEvent evt) {  System.*out*.println("把Person对象存放到session中：" + evt.getValue());  }  **public** **void** valueUnbound(HttpSessionBindingEvent evt) {  System.*out*.println("从session中移除Pseron对象：" + evt.getValue());  }  } |

ListenerServlet.java

|  |
| --- |
| **public** **class** ListenerServlet **extends** BaseServlet {  **public** String addPerson(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  Person p = **new** Person("zhangSan", 23, "male");  request.getSession().setAttribute("person", p);  **return** "/index.jsp";  }    **public** String removePerson(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.getSession().removeAttribute("person");  **return** "/index.jsp";  } |

index.jsp

|  |
| --- |
| <body>  <a href=*"*<c:url value=*'/ListenerServlet?method=addPerson'*/>*"*>addPerson</a>  <br/>  <a href=*"*<c:url value=*'/ListenerServlet?method=removePerson'*/>*"*>removePerson</a>  <br/>  </body> |

* HttpSessionActivationListener：Tomcat会在session从时间不被使用时钝化session对象，所谓钝化session，就是把session通过序列化的方式保存到硬盘文件中。当用户再使用session时，Tomcat还会把钝化的对象再活化session，所谓活化就是把硬盘文件中的session在反序列化回内存。当session被Tomcat钝化时，session中存储的对象也被纯化，当session被活化时，也会把session中存储的对象活化。如果某个类实现了HttpSessionActiveationListener接口后，当对象随着session被钝化和活化时，下面两个方法就会被调用：
* public void sessionWillPassivate([HttpSessionEvent](mk:@MSITStore:F:\帮助\J2EE1.5.chm::/javax/servlet/http/HttpSessionEvent.html) se)：当对象感知被活化时调用本方法；
* public void sessionDidActivate([HttpSessionEvent](mk:@MSITStore:F:\帮助\J2EE1.5.chm::/javax/servlet/http/HttpSessionEvent.html) se)：当对象感知被钝化时调用本方法；

HttpSessionActivationListener监听器与HttpSessionBindingListener监听器相似，都是感知型的监听器，例如让Person类实现了HttpSessionActivationListener监听器接口，并把Person对象添加到了session中后，当Tomcat钝化session时，同时也会钝化session中的Person对象，这时Person对象就会感知到自己被钝化了，其实就是调用Person对象的sessionWillPassivate()方法。当用户再次使用session时，Tomcat会活化session，这时Person会感知到自己被活化，其实就是调用Person对象的sessionDidActivate()方法。

注意，因为钝化和活化session，其实就是使用序列化和反序列化技术把session从内存保存到硬盘，和把session从硬盘加载到内存。这说明如果Person类没有实现Serializable接口，那么当session钝化时就不会钝化Person，而是把Person从session中移除再钝化！这也说明session活化后，session中就不在有Person对象了。

示例步骤：

* 先不管HttpSessionActivationListener监听器接口，先来配置Tomcat钝化session的参数，把下面配置文件放到tomcat\conf\catalina\localhost目录下！文件名称为项目名称。

|  |
| --- |
| <Context>  <Manager className=*"org.apache.catalina.session.PersistentManager"* maxIdleSwap=*"1"*>  <Store className=*"org.apache.catalina.session.FileStore"* directory=*"mysession"*/>  </Manager>  </Context> |

　　　　访问项目的index.jsp页面，这会使Tomcat创建Session对象，然后等待一分钟后，查看Tomcat\work\Catalina\localhost\listener\mysession目录下是否会产生文件，如果产生了，说明钝化session的配置成功了，可以开始下一步了。

* 创建Person类，让Person类实现HttpSessionActivationListener和Serializable接口：

Person.java

|  |
| --- |
| **public** **class** Person **implements** HttpSessionActivationListener, Serializable {  **private** String name;  **private** **int** age;  **private** String sex;    **public** Person(String name, **int** age, String sex) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.sex = sex;  }  **public** Person() {  **super**();  }  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** String getSex() {  **return** sex;  }  **public** **void** setSex(String sex) {  **this**.sex = sex;  }  **public** **void** sessionDidActivate(HttpSessionEvent evt) {  System.*out*.println("session已经活化");  }  **public** **void** sessionWillPassivate(HttpSessionEvent evt) {  System.*out*.println("session被钝化了！");  }  } |

* 与上例一样，编写Servlet，提供两个方法：一个向session中添加Person对象，另一个从session中移除Person对象：

Person.java

|  |
| --- |
| **public** **class** ListenerServlet **extends** BaseServlet {  **public** String addPerson(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  Person p = **new** Person("zhangSan", 23, "male");  request.getSession().setAttribute("person", p);  **return** "/index.jsp";  }    **public** String removePerson(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.getSession().removeAttribute("person");  **return** "/index.jsp";  }  } |

* 在index.jsp页面中给出访问addPerson()和removePerson()的方法：

|  |
| --- |
| <body>  <a href=*"*<c:url value=*'/ListenerServlet?method=addPerson'*/>*"*>addPerson</a>  <br/>  <a href=*"*<c:url value=*'/ListenerServlet?method=removePerson'*/>*"*>removePerson</a>  <br/>  </body> |

* 打开index.jsp页面，这时Tomcat会创建session，必须在1分钟之前点击addPerson链接，这能保证在session被钝化之前把Person对象添加到session中；
* 等待一分钟，这时session会被钝化，也就会调用Person的sessionWillPassivate()；
* 刷新一下index.jsp页面，这会使session活化，会调用Person的sessionDidActivate()方法。
* 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

## 在线人数统计

# 过滤器（Filter）

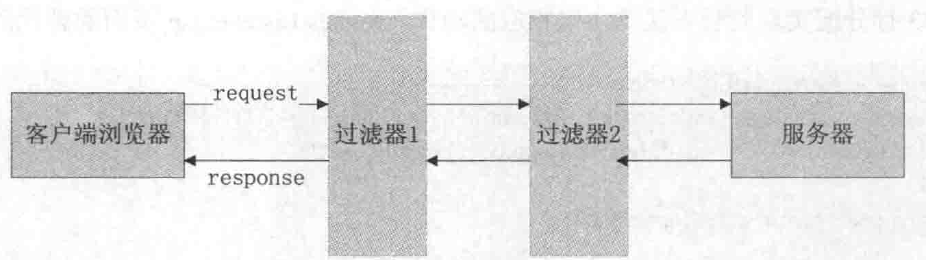
## 过滤器概述

javaweb的三大组件：Servlet，Listener(两个感知监听器不需要配置)，Filter

### 什么是过滤器

过滤器JavaWeb三大组件之一，它与Servlet很相似！不它过滤器是用来拦截请求的，而不是处理请求的。Java三大器：过滤器， 监听器，拦截器(springmvc,struts2框架里的)。

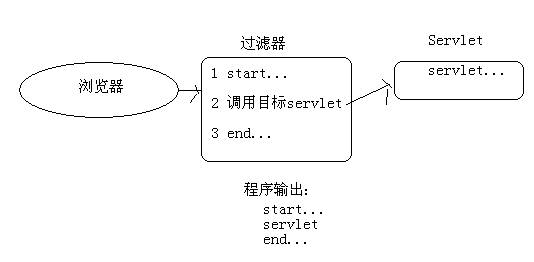
何为过滤器?顾名思义，它的作用就是阻挡某些事件的发生。在Web应用程序中，过滤器是介于Servlet之前，既可以拦截、过滤浏览器的请求，也可以改变对浏览器的响应。它在服务器端与客户端起到了一个中间组件的作用，对二者之间的数据信息进行过滤，其处理过程如图所示。由图看出，当客户端浏览器发起一个请求时，服务器端的过滤器将检查请求数据中的内容，它可改变这些内容或者重新设置报头信息，再转发给服务器上被请求的目标资源，处理完毕后再向客户端响应处理结果。过滤的效果是双向的。



一个Web应用程序，可以有多个过滤器，组成一个过滤器链，如经常使用过滤器完成字符编码的设定和验证用户的合法性。过滤器链中的每个过滤器都各司其职地处理并转发数据。

当用户请求某个Servlet时，会先执行部署在这个请求上的Filter，如果Filter“放行”，那么会继承执行用户请求的Servlet；如果Filter不“放行”，那么就不会执行用户请求的Servlet。

其实可以这样理解，当用户请求某个Servlet时，Tomcat会去执行注册在这个请求上的Filter，然后是否“放行”由Filter来决定。可以理解为，Filter来决定是否调用Servlet！当执行完成Servlet的代码后，还会执行Filter后面的代码。



### 过滤器的功能

1对用户请求进行身份认证。

2对用户发送的数据进行过滤或者替换。

3转换图像的数据格式。

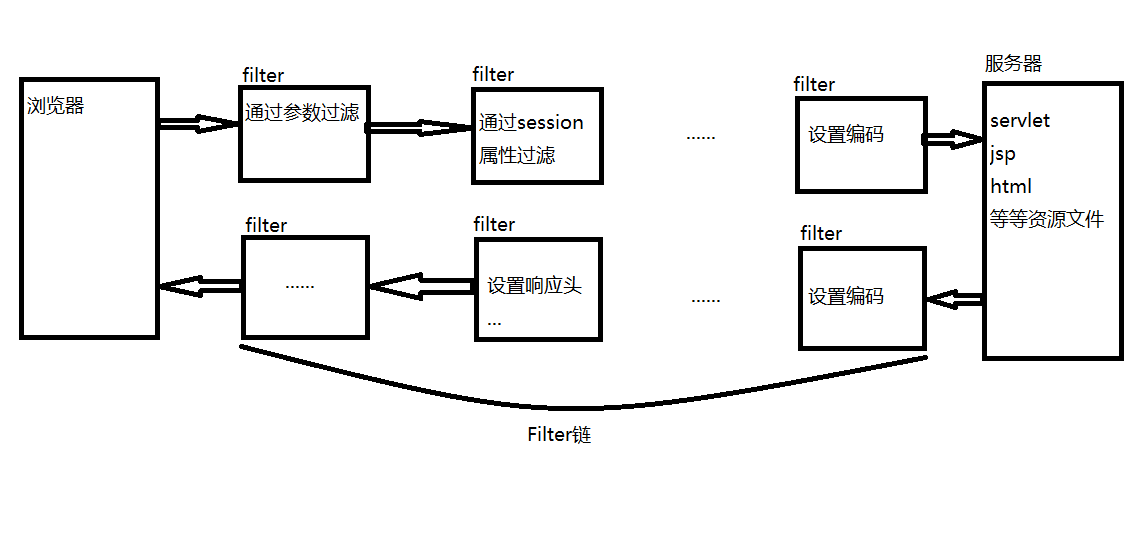
4数据压缩。

5数据加密。

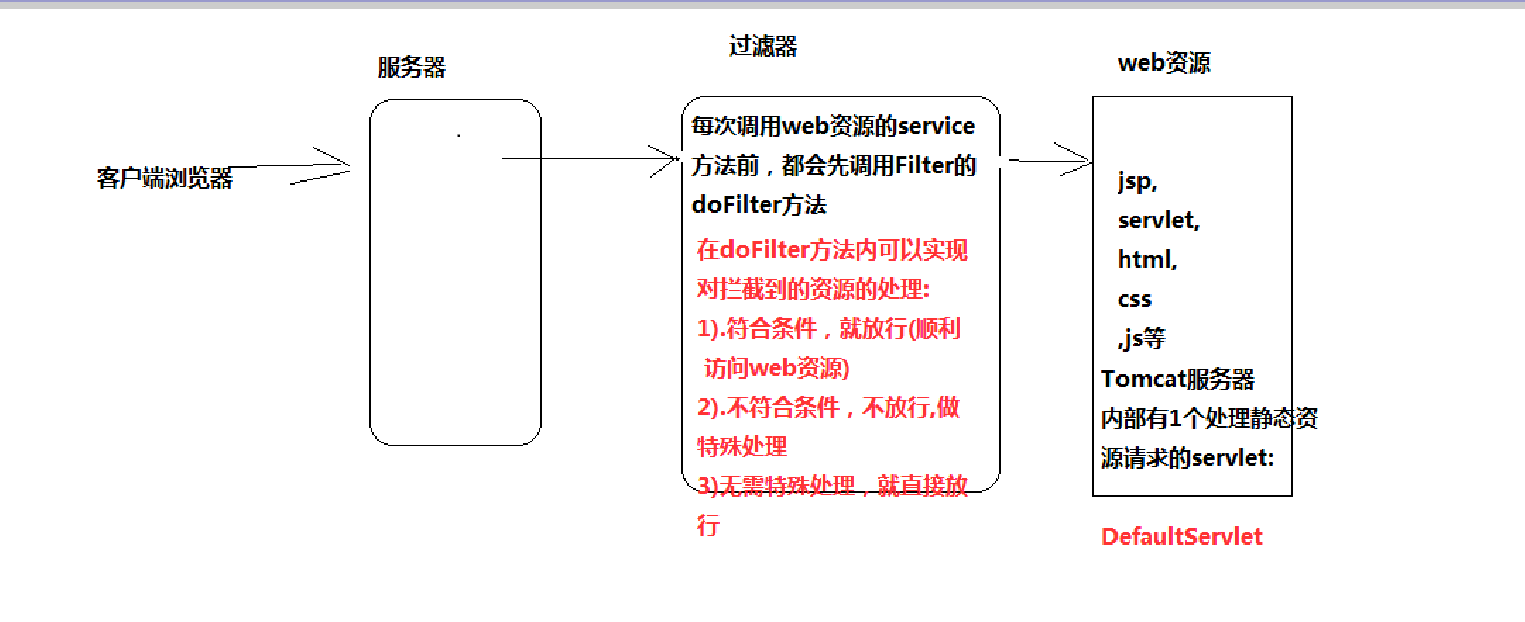
6XML数据的转换。

7修改请求数据的字符集。

8拦截ip



浏览器发出的请求先递交给第一个filter进行过滤，符合规则则放行，递交给filter链中的下一个过滤器进行过滤。过滤器在链中的顺序与它在web.xml中配置的顺序有关，配置在前的则位于链的前端。当请求通过了链中所有过滤器后就可以访问资源文件了，如果不能通过，则可能在中间某个过滤器中被处理掉。



## 过滤器之hello world

　　其实过滤器与Servlet很相似，我们回忆一下如果写的第一个Servlet应用！写一个类，实现Servlet接口！没错，写过滤器就是写一个类，实现Filter接口。

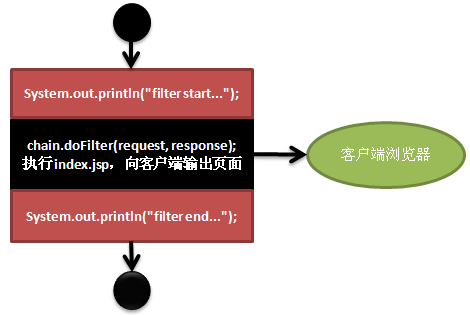
|  |
| --- |
| **1、实现拦截器，首先要将类继承Filter接口。**  **public** **class** HelloFilter **implements** Filter {  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {}  2、重写过滤器的doFilter(request，response，chain)方法。另外两个init()、destroy()方法一般不需要重写。在doFilter方法中进行过滤操作。  常用代码有：获取请求、获取响应、获取session、放行。  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  3、对请求数据操作。  System.*out*.println("Hello Filter");  4、调用chain对象的doFilter方法，可以释放拦截，可以访问到目标资源。  chain.doFilter(request.response)//  5、对响应数据进行操作。  }  **public** **void** destroy() {//这里一般不写代码 }  } |

第二步也与Servlet一样，在web.xml文件中部署Filter：

|  |
| --- |
| 1、配置过滤器的别名和全限定类名。  <filter>  <filter-name>helloFilter</filter-name>  <filter-class>cn.itcast.filter.HelloFilter</filter-class>  </filter>  2、配置过滤器所拦截的资源。  <filter-mapping>  <filter-name>helloFilter</filter-name>  <url-pattern>/index.jsp</url-pattern>”/\*” 表示拦截该项目的所有资源。  </filter-mapping> |

当用户访问index.jsp页面时，会执行HelloFilter的doFilter()方法！在我们的示例中，index.jsp页面是不会被执行的，如果想执行index.jsp页面，那么我们需要放行！

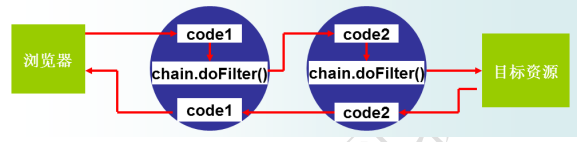
|  |
| --- |
| **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("filter start...");  chain.doFilter(request, response);  System.*out*.println("filter end...");  } |



　　有很多同学总是错误的认为，一个请求在给客户端输出之后就算是结束了，这是不对的！其实很多事情都需要在给客户端响应之后才能完成！

### Filter的执行顺序

在doFilter()方法中，chain.doFilter()前的一般是对request执行的过滤操作，chain.doFilter后面的代码一般是对response执行的操作。



## Fliter类（javax.servlet.Filter）

### 过滤器的生命周期

我们已经学习过Servlet的生命周期，那么Filter的生命周期也就没有什么难度了！

Fileter接口（三个方法）

init(FilterConfig)：在服务器启动时会创建Filter实例，并且每个类型的Filter只创建一个实例，从此不再创建！在创建完Filter实例后，会马上调用init()方法完成初始化工作，这个方法只会被执行一次。filterConfig参数是一个FilterConfig对象。利用该对象可以得到过滤器中初始化的配置参数信息。如果一个目标资源有多个Filter，那么init()也只会执行一次。

doFilter(ServletRequest req,ServletResponse res,FilterChain chain)：方法是过滤器中主要实现过滤的方法。当客户端请求目标资源时，Web应用程序会调用与此目标资源相关的doFilter()方法，在该方法中，实现对请求和响应的数据处理。参数request表示客户端的请求，response表示对应请求的响应，chain是过滤器链对象。在方法中的特定操作完成后，可调用FilterChain对象的doFilter(request,response)将请求传给过滤器链中的下一个过滤器，也可以直接返回响应内容，还可以将目标重定向。

destroy()：服务器会在创建Filter对象之后，把Filter放到缓存中一直使用，通常不会销毁它。一般会在服务器关闭时销毁Filter对象，在销毁Filter对象之前，服务器会调用Filter对象的destory()方法。

### FilterConfig

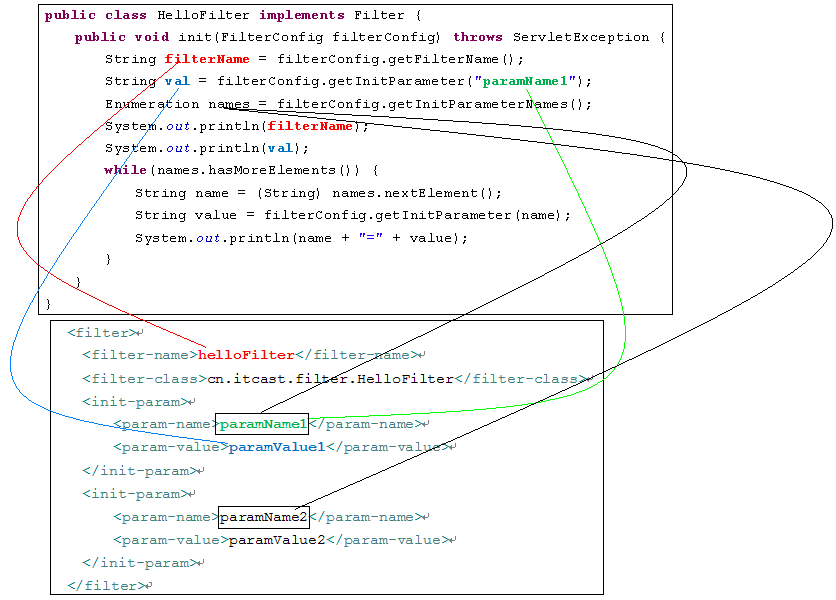
该接口用于在过滤器初始化时由Web容器向过滤器传送初始化配置参数，并传入到过滤器对象的init()方法中。FilterConfig接口中有4个方法可以调用。下面是FilterConfig的功能介绍：

ServletContext getServletContext()：获取ServletContext的方法；

String getFilterName()：获取Filter的配置名称；与<filter-name>元素对应；

String getInitParameter(String name)：获取Filter的初始化配置，与<init-param>元素对应；

Enumeration getInitParameterNames()：获取所有初始化参数的名称。



### FilterChain

doFilter()方法的参数中有一个类型为FilterChain的参数，它只有一个方法：doFilter(ServletRequest,ServletResponse)。

前面我们说doFilter()方法的放行，让请求流访问目标资源！但这么说不严密，其实调用该方法的意思是，“我（当前Filter）”放行了，但不代表其他人（其他过滤器）也放行。

也就是说，一个目标资源上，可能部署了多个过滤器，就好比在你去北京的路上有多个打劫的匪人（过滤器），而其中第一伙匪人放行了，但不代表第二伙匪人也放行了，所以调用FilterChain类的doFilter()方法表示的是执行下一个过滤器的doFilter()方法，或者是执行目标资源！

如果当前过滤器是最后一个过滤器，那么调用chain.doFilter()方法表示执行目标资源，而不是最后一个过滤器，那么chain.doFilter()表示执行下一个过滤器的doFilter()方法。

放行，相当于调用了servlet的service方法。执行之后完之后，执行之后的代码。

## 设置过滤器

### web.xml中配置

|  |
| --- |
| 1、注册filter  <filter>  <description>FilterDemo </description>  <filter-name>FilterDemo02</filter-name>  <filter-class>me.gacl.web.filter.FilterDemo02</filter-class>  <!--配置FilterDemo02过滤器的初始化参数-->  <init-param>  <description>配置FilterDemo02过滤器的初始化参数</description>  <param-name>name</param-name>  <param-value>gacl</param-value>  </init-param>  <init-param>  <description>配置FilterDemo02过滤器的初始化参数</description>  <param-name>like</param-name>  <param-value>java</param-value>  </init-param>  </filter>  <description>用于添加描述信息，该元素的内容可为空，<description>可以不配置。  <filter-name>用于为过滤器指定一个名字，该元素的内容不能为空。  <filter-class>元素用于指定过滤器的完整的限定类名。  <init-param>元素用于为过滤器指定初始化参数，它的子元素<param-name>指定参数的名字，<param-value>指定参数的值。在过滤器中，可以使用FilterConfig接口对象来访问初始化参数。如果过滤器不需要指定初始化参数，那么<init-param>元素可以不配置。  2、映射Filter：在web.xml文件中注册了Filter之后，还要在web.xml文件中映射Filter  <!--映射过滤器-->  <filter-mapping>  <filter-name>FilterDemo02</filter-name>  <!--“/\*”表示拦截所有的请求 -->  <url-pattern>/\*</url-pattern>  </filter-mapping>‘’  <filter-mapping>元素用于设置一个 Filter 所负责拦截的资源。一个Filter拦截的资源可通过两种方式来指定：Servlet 名称和资源访问的请求路径  <filter-name>子元素用于设置filter的注册名称。该值必须是在<filter>元素中声明过的过滤器的名字  <url-pattern>设置 filter 所拦截的请求路径(过滤器关联的URL样式)  <servlet-name>指定过滤器所拦截的Servlet名称。  <dispatcher>指定过滤器所拦截的资源被 Servlet 容器调用的方式，可以是REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST。用户可以设置多个<dispatcher> 子元素用来指定 Filter 对资源的多种调用方式进行拦截。 |

### url-pattern路径

1)精确拦截：指拦截某个具体的资源，如：/view/addComment.jsp

2)模糊拦截：包含\*,又分为拦截所有,后缀名拦截和目录拦截。

拦截所有，即: /\* (在doFilter方法中需要对特殊的资源进行放行)

后缀名拦截，即\*.后缀名，如\*.jsp，是拦截所有jsp页面

目录拦截，如：/view/\*,拦截View目录下所有资源

注：

a)url-pattern要么以斜杠开头，要么以\*开头 例如： /hello.action或\*.jsp

b)不能同时使用两个模糊拦截。例如 /\*.do 是非法的

## 多个过滤器执行顺序

### 过滤器的顺序问题

一个目标资源可以指定多个过滤器，过滤器的执行顺序是在web.xml文件中的部署顺序：

|  |
| --- |
| <filter>1、首先执行myFilter1  <filter-name>myFilter1</filter-name>  <filter-class>cn.itcast.filter.MyFilter1</filter-class>  </filter>  <filter-mapping>  <filter-name>myFilter1</filter-name>  <url-pattern>/index.jsp</url-pattern>//访问的静态资源或动态资源相同  //如果这里没有设置index.jsp路径，那么访问时，还是会转到主页，而不经过过滤器。  </filter-mapping>  <filter>2、然后执行myFilter2  <filter-name>myFilter2</filter-name>  <filter-class>cn.itcast.filter.MyFilter2</filter-class>  </filter>  <filter-mapping>  <filter-name>myFilter2</filter-name>  <url-pattern>/index.jsp</url-pattern>//访问的静态资源或动态资源相同  </filter-mapping> |
| **public** **class** MyFilter1 **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request, HttpServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("filter1 start...");  chain.doFilter(request, response);//放行，执行MyFilter2的doFilter()方法  System.*out*.println("filter1 end...");  }  } |
| **public** **class** MyFilter2 **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request, HttpServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("filter2 start...");  chain.doFilter(request, response);//放行，执行目标资源  System.*out*.println("filter2 end...");  }  } |
| <body>  This is my JSP page. <br>  <h1>index.jsp</h1>  <%System.out.println("index.jsp"); %>  </body> |

当有用户访问index.jsp页面时，输出结果如下：

|  |
| --- |
| filter1 start...  filter2 start...  index.jsp  filter2 end...  filter1 end... |

## 四种拦截方式

我们来做个测试，写一个过滤器，指定过滤的资源为b.jsp，然后我们在浏览器中直接访问b.jsp，你会发现过滤器执行了！

但是，当我们在a.jsp中request.getRequestDispathcer(“/b.jsp”).forward(request,response)时，就不会再执行过滤器了！也就是说，默认情况下，只能直接访问目标资源才会执行过滤器，而forward执行目标资源，不会执行过滤器！

|  |
| --- |
| **public** **class** MyFilter **extends** HttpFilter {  **public** **void** doFilter(HttpServletRequest request,  HttpServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  System.*out*.println("myfilter...");  chain.doFilter(request, response);  }  } |
| <filter>  <filter-name>myfilter</filter-name>  <filter-class>cn.itcast.filter.MyFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  </filter-mapping> |
| <body>  <h1>b.jsp</h1>  </body> |
| <h1>a.jsp</h1>  <%  request.getRequestDispatcher("/b.jsp").forward(request, response);  %>  </body> |

http://localhost:8080/filtertest/b.jsp -->直接访问b.jsp时，会执行过滤器内容；

http://localhost:8080/filtertest/a.jsp --> 访问a.jsp，但a.jsp会forward到b.jsp，这时就不会执行过滤器！

其实过滤器有四种拦截方式！分别是：REQUEST、FORWARD、INCLUDE、ERROR。

* REQUEST：直接访问目标资源时执行过滤器。包括：在地址栏中直接访问、表单提交、超链接、重定向，只要在地址栏中可以看到目标资源的路径，就是REQUEST；
* FORWARD：转发访问执行过滤器。包括RequestDispatcher#forward()方法、<jsp:forward>标签都是转发访问；
* INCLUDE：包含访问执行过滤器。包括RequestDispatcher#include()方法、<jsp:include>标签都是包含访问；
* ERROR：当目标资源在web.xml中配置为<error-page>中时，并且真的出现了异常，转发到目标资源时，会执行过滤器。

可以在<filter-mapping>中添加0~n个<dispatcher>子元素，来说明当前访问的拦截方式。

|  |
| --- |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  <dispatcher>REQUEST</dispatcher>  <dispatcher>FORWARD</dispatcher>  </filter-mapping> |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  </filter-mapping> |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  <dispatcher>FORWARD</dispatcher>  </filter-mapping> |

其实最为常用的就是REQUEST和FORWARD两种拦截方式，而INCLUDE和ERROR都比较少用！其中INCLUDE比较好理解，我们这里不再给出代码，学员可以通过FORWARD方式修改，来自己测试。而ERROR方式不易理解，下面给出ERROR拦截方式的例子：

|  |
| --- |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/b.jsp</url-pattern>  <dispatcher>ERROR</dispatcher>  </filter-mapping>  <error-page>  <error-code>500</error-code>  <location>/b.jsp</location>  </error-page> |
| <body>  <h1>a.jsp</h1>  <%  **if**(**true**)  **throw** **new** RuntimeException("嘻嘻~");  %>  </body> |

## 过滤器的应用场景

过滤器的应用场景：

执行目标资源之前做**预处理**工作，例如设置编码，这种试**通常都会放行**，只是在目标资源执行之前做一些准备工作；

通过条件判断是否放行，例如校验当前用户是否已经登录，或者用户IP是否已经被禁用；

在目标资源执行后，做一些后续的特殊处理工作，例如把目标资源输出的数据进行处理；

### 设置编码(EncodingFilter)

|  |
| --- |
| public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  1、强转为ServletRequest  HttpServletRequest request=(HttpServletRequest) request;  HttpServletResponse response=(HttpServletResponse) response;  2、设置请求编码格式  request.setCharacterEncoding("UTF-8");  3、设置响应编码格式  response.setContentType(“text/html;charset=utf-8”) ;  4、调用doFilter，调用其他的Filter或者是目标资源。子接口或者是父接口对象都可以。  chain.doFilter(request, response);  } |
| <servlet>  <servlet-name>myservlet</servlet-name>  <servlet-class>cn.itcast.servlet.MyServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>myservlet</servlet-name>  <url-pattern>/\*</url-pattern>//这里使用统配符\*，不管是静态资源还是动态资源，都可以设置编码格式。  </servlet-mapping> |

### 登录权限验证过滤器

假设系统中某些页面不能够直接访问，需要先登录餐能访问，规定如下两类页面未登录不能直接访问：

1)：.web隐痛根路径下的view目录里的jsp页面。

2：WEB-INF目录里的页面。

开发一个登陆权限验证过滤器，如果没有登陆，那么不能够访问以上两类页面。

设置xml

|  |
| --- |
| <!-- 登录权限验证过滤器 -->  <filter>  <filter-name>loginFilter</filter-name>  <filter-class>cn.sxt.filter.LoginFilter</filter-class>  <!-- 把需要直接放行的特殊资源配置成初始化参数 -->  <init-param>  <param-name>passResource</param-name>  <!-- 把直接放行的资源配置成初始化参数的值-->  <param-value>login.action,authCode.action,.png</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>loginFilter</filter-name>  <url-pattern>\*.action</url-pattern>  <url-pattern>/view/\*</url-pattern>  </filter-mapping> |

代码

|  |
| --- |
|  |

## 设置目标资源

在web.xml文件中部署Filter时，可以通过“\*”来执行目标资源：

|  |
| --- |
| <filter-mapping>  <filter-name>myfilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

这一特性与Servlet完全相同！通过这一特性，我们可以在用户访问敏感资源时，执行过滤器，例如：<url-pattern>/admin/\*<url-pattern>，可以把所有管理员才能访问的资源放到/admin路径下，这时可以通过过滤器来校验用户身份。

还可以为<filter-mapping>指定目标资源为某个Servlet，例如：

|  |
| --- |
| <servlet>  <servlet-name>**myservlet**</servlet-name>  <servlet-class>cn.itcast.servlet.MyServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>**myservlet**</servlet-name>  <url-pattern>/abc</url-pattern>  </servlet-mapping>  <filter>  <filter-name>myfilter</filter-name>  <filter-class>cn.itcast.filter.MyFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>myfilter</filter-name>  <servlet-name>**myservlet**</servlet-name>  </filter-mapping> |

　　当用户访问http://localhost:8080/filtertest/abc时，会执行名字为myservlet的Servlet，这时会执行过滤器。

## Filter小结

Filter的三个方法：

* void init(FilterConfig)：在Tomcat启动时被调用；
* void destroy()：在Tomcat关闭时被调用；
* void doFilter(ServletRequest,ServletResponse,FilterChain)：每次有请求时都调用该方法；

FilterConfig类：与ServletConfig相似，用来获取Filter的初始化参数

* ServletContext getServletContext()：获取ServletContext的方法；
* String getFilterName()：获取Filter的配置名称；
* String getInitParameter(String name)：获取Filter的初始化配置，与<init-param>元素对应；
* Enumeration getInitParameterNames()：获取所有初始化参数的名称。

FilterChain类：

* void doFilter(ServletRequest,ServletResponse)：放行！表示执行下一个过滤器，或者执行目标资源。可以在调用FilterChain的doFilter()方法的前后添加语句，在FilterChain的doFilter()方法之前的语句会在目标资源执行之前执行，在FilterChain的doFilter()方法之后的语句会在目标资源执行之后执行。

四各拦截方式：REQUEST、FORWARD、INCLUDE、ERROR，默认是REQUEST方式。

* REQUEST：拦截直接请求方式；
* FORWARD：拦截请求转发方式；
* INCLUDE：拦截请求包含方式；
* ERROR：拦截错误转发方式。

过滤器特点及与今后拦截器的区别：

过滤器是基于URL的拦截，它会对控制器中的所有方法都拦截；

拦截器是基于方法的拦截，它可以明确去拦截控制器指定的方法法

# 文件上传

## 文件上传概述

### 文件上传的作用

例如网络硬盘！就是用来上传下载文件的。

### 文件上传对页面的要求

上传文件的要求比较多，需要记一下：

1. 必须使用表单，而不能是超链接；
2. 表单的method必须是POST，而不能是GET；
3. 表单的enctype必须是multipart/form-data；
4. 在表单中添加file表单字段，即<input type=”file”…/>

|  |
| --- |
| <form action=*"*${pageContext.request.contextPath }*/FileUploadServlet"* method=*"post"* enctype=*"multipart/form-data"*>  用户名：<input type=*"text"* name=*"username"*/><br/>  文件1：<input type=*"file"* name=*"file1"*/><br/>  文件2：<input type=*"file"* name=*"file2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |

### 比对文件上传表单和普通文本表单的区别

通过httpWatch查看“文件上传表单”和“普通文本表单”的区别。

文件上传表单的enctype=”multipart/form-data”，表示多部件表单数据；

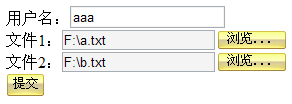
普通文本表单可以不设置enctype属性：

当method=”post”时，enctype的默认值为application/x-www-form-urlencoded，表示使用url编码正文；

当method=”get”时，enctype的默认值为null，没有正文，所以就不需要enctype了。

**对普通文本表单的测试**：

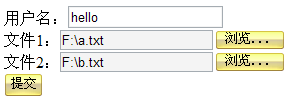
|  |
| --- |
| <form action=*"*${pageContext.request.contextPath }*/FileUploadServlet"* method=*"post"*>  用户名：<input type=*"text"* name=*"username"*/><br/>  文件1：<input type=*"file"* name=*"file1"*/><br/>  文件2：<input type=*"file"* name=*"file2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |



　　通过httpWatch测试，查看表单的请求数据正文，我们发现请求中只有文件名称，而没有文件内容。也就是说，当表单的enctype不是multipart/form-data时，请求中不包含文件内容，而只有文件的名称，这说明普通文本表单中input:file与input:text没什么区别了。

**对文件上传表单的测试**：

|  |
| --- |
| <form action=*"*${pageContext.request.contextPath }*/FileUploadServlet"* method=*"post"* enctype=*"multipart/form-data"*>  用户名：<input type=*"text"* name=*"username"*/><br/>  文件1：<input type=*"file"* name=*"file1"*/><br/>  文件2：<input type=*"file"* name=*"file2"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |





通过httpWatch测试，查看表单的请求数据正文部分，发现正文部分是由多个部件组成，每个部件对应一个表单字段，每个部件都有自己的头信息。头信息下面是空行，空行下面是字段的正文部分。多个部件之间使用随机生成的分隔线隔开。

文本字段的头信息中只包含一条头信息，即Content-Disposition，这个头信息的值有两个部分，第一部分是固定的，即form-data，第二部分为字段的名称。在空行后面就是正文部分了，正文部分就是在文本框中填写的内容。

　　文件字段的头信息中包含两条头信息，Content-Disposition和Content-Type。Content-Disposition中多出一个filename，它指定的是上传的文件名称。而Content-Type指定的是上传文件的类型。文件字段的正文部分就是文件的内容。

　　请注意，因为我们上传的文件都是普通文本文件，即txt文件，所以在httpWatch中是可以正常显示的，如果上传的是exe、mp3等文件，那么在httpWatch看到的就是乱码了。

### 文件上传对Servlet的要求

当提交的表单是文件上传表单时，那么对Servlet也是有要求的。

首先我们要肯定一点，文件上传表单的数据也是被封装到request对象中的。

**request.getParameter(String)方法获取指定的表单字段字符内容，但文件上传表单已经不在是字符内容，而是字节内容，所以失效。**

这时可以使用request的getInputStream()方法获取ServletInputStream对象，它是InputStream的子类，这个ServletInputStream对象对应整个表单的正文部分（从第一个分隔线开始，到最后），这说明我们需要的解析流中的数据。当然解析它是很麻烦的一件事情，而Apache已经帮我们提供了解析它的工具：commons-fileupload。

　　可以尝试把request.getInputStream()这个流中的内容打印出来，再对比httpWatch中的请求数据。

|  |
| --- |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  InputStream in = request.getInputStream();  String s = IOUtils.*toString*(in);  System.*out*.println(s);  } |
| -----------------------------7ddd3370ab2  Content-Disposition: form-data; name="username"  hello  -----------------------------7ddd3370ab2  Content-Disposition: form-data; name="file1"; filename="a.txt"  Content-Type: text/plain  aaa  -----------------------------7ddd3370ab2  Content-Disposition: form-data; name="file2"; filename="b.txt"  Content-Type: text/plain  bbb  -----------------------------7ddd3370ab2-- |

## commons-fileupload

为什么使用fileupload：

上传文件的要求比较多，需要记一下：

* 必须是POST表单；
* 表单的enctype必须是multipart/form-data；
* 在表单中添加file表单字段，即<input type=”file”…/>

Servlet的要求：

* 不能再使用request.getParameter()来获取表单数据；
* 可以使用request.getInputStream()得到所有的表单数据，而不是一个表单项的数据；
* 这说明不使用fileupload，我们需要自己来对request.getInputStream()的内容进行解析！！！

### fileupload概述

fileupload是由apache的commons组件提供的上传组件。它最主要的工作就是帮我们解析request.getInputStream()。

fileupload组件需要的JAR包有：

* commons-fileupload.jar，核心包；
* commons-io.jar，依赖包。

### fileupload简单应用

　　fileupload的核心类有：DiskFileItemFactory、ServletFileUpload、FileItem。

使用fileupload组件的步骤如下：

1. 创建工厂类DiskFileItemFactory对象：DiskFileItemFactory factory = new DiskFileItemFactory()
2. 使用工厂创建解析器对象：ServletFileUpload fileUpload = new ServletFileUpload(factory)
3. 使用解析器来解析request对象：List<FileItem> list = fileUpload.parseRequest(request)

隆重介绍FileItem类，它才是我们最终要的结果。一个FileItem对象对应一个表单项（表单字段）。一个表单中存在文件字段和普通字段，可以使用FileItem类的isFormField()方法来判断表单字段是否为普通字段，如果不是普通字段，那么就是文件字段了。

* String getName()：获取文件字段的文件名称；
* String getString()：获取字段的内容，如果是文件字段，那么获取的是文件内容，当然上传的文件必须是文本文件；
* String getFieldName()：获取字段名称，例如：<input type=”text” name=”username”/>，返回的是username；
* String getContentType()：获取上传的文件的类型，例如：text/plain。
* int getSize()：获取上传文件的大小；
* boolean isFormField()：判断当前表单字段是否为普通文本字段，如果返回false，说明是文件字段；
* InputStream getInputStream()：获取上传文件对应的输入流；
* void write(File)：把上传的文件保存到指定文件中。

### 简单上传示例

写一个简单的上传示例：

* 表单包含一个用户名字段，以及一个文件字段；
* Servlet保存上传的文件到uploads目录，显示用户名，文件名，文件大小，文件类型。

第一步：

完成index.jsp，只需要一个表单。注意表单必须是post的，而且enctype必须是mulitpart/form-data的。

|  |
| --- |
| <form action=*"*${pageContext.request.contextPath }*/FileUploadServlet"* method=*"post"* enctype=*"multipart/form-data"*>  用户名：<input type=*"text"* name=*"username"*/><br/>  文件1：<input type=*"file"* name=*"file1"*/><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |

第二步：

完成FileUploadServlet

|  |
| --- |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  // 因为要使用response打印，所以设置其编码  response.setContentType("text/html;charset=utf-8");    // 创建工厂  DiskFileItemFactory dfif = **new** DiskFileItemFactory();  // 使用工厂创建解析器对象  ServletFileUpload fileUpload = **new** ServletFileUpload(dfif);  **try** {  // 使用解析器对象解析request，得到FileItem列表  List<FileItem> list = fileUpload.parseRequest(request);  // 遍历所有表单项  **for**(FileItem fileItem : list) {  // 如果当前表单项为普通表单项  **if**(fileItem.isFormField()) {  // 获取当前表单项的字段名称  String fieldName = fileItem.getFieldName();  // 如果当前表单项的字段名为username  **if**(fieldName.equals("username")) {  // 打印当前表单项的内容，即用户在username表单项中输入的内容  response.getWriter().print("用户名：" + fileItem.getString() + "<br/>");  }  } **else** {//如果当前表单项不是普通表单项，说明就是文件字段  String name = fileItem.getName();//获取上传文件的名称  // 如果上传的文件名称为空，即没有指定上传文件  **if**(name == **null** || name.isEmpty()) {  **continue**;  }  // 获取真实路径，对应${项目目录}/uploads，当然，这个目录必须存在  String savepath = **this**.getServletContext().getRealPath("/uploads");  // 通过uploads目录和文件名称来创建File对象  File file = **new** File(savepath, name);  // 把上传文件保存到指定位置  fileItem.write(file);  // 打印上传文件的名称  response.getWriter().print("上传文件名：" + name + "<br/>");  // 打印上传文件的大小  response.getWriter().print("上传文件大小：" + fileItem.getSize() + "<br/>");  // 打印上传文件的类型  response.getWriter().print("上传文件类型：" + fileItem.getContentType() + "<br/>");  }  }  } **catch** (Exception e) {  **throw** **new** ServletException(e);  }  } |

## 文件上传之细节

### 把上传的文件放到WEB-INF目录下

如果没有把用户上传的文件存放到WEB-INF目录下，那么用户就可以通过浏览器直接访问上传的文件，这是非常危险的。

假如说用户上传了一个a.jsp文件，然后用户在通过浏览器去访问这个a.jsp文件，那么就会执行a.jsp中的内容，如果在a.jsp中有如下语句：Runtime.getRuntime().exec(“shutdown –s –t 1”);，那么你就会…

通常我们会在WEB-INF目录下创建一个uploads目录来存放上传的文件，而在Servlet中找到这个目录需要使用ServletContext的getRealPath(String)方法，例如在我的upload1项目中有如下语句：

ServletContext servletContext = this.getServletContext();

String savepath = servletContext.getRealPath(“/WEB-INF/uploads”);

其中savepath为：F:\tomcat6\_1\webapps\upload1\WEB-INF\uploads。

### 文件名称（完整路径、文件名称）

**上传文件名称可能是完整路径**：

IE6获取的上传文件名称是完整路径，而其他浏览器获取的上传文件名称只是文件名称而已。浏览器差异的问题我们还是需要处理一下的。

|  |
| --- |
| String name = file1FileItem.getName();  response.getWriter().print(name); |

使用不同浏览器测试，其中IE6就会返回上传文件的完整路径，不知道IE6在搞什么，这给我们带来了很大的麻烦，就是需要处理这一问题。

处理这一问题也很简单，无论是否为完整路径，我们都去截取最后一个“\\”后面的内容就可以了。

|  |
| --- |
| String name = file1FileItem.getName();  **int** lastIndex = name.lastIndexOf("\\");//获取最后一个“\”的位置  **if**(lastIndex != -1) {//注意，如果不是完整路径，那么就不会有“\”的存在。  name = name.substring(lastIndex + 1);//获取文件名称  }  response.getWriter().print(name); |

### 中文乱码问题

**上传文件名称中包含中文**：

当上传的谁的名称中包含中文时，需要设置编码，commons-fileupload组件为我们提供了两种设置编码的方式：

* request.setCharacterEncoding(String)：这种方式是我们最为熟悉的方式了；
* fileUpload.setHeaderEncdoing(String)：这种方式的优先级高与前一种。

**上传文件的文件内容包含中文：**

通常我们不需关心上传文件的内容，因为我们会把上传文件保存到硬盘上！也就是说，文件原来是什么样子，到服务器这边还是什么样子！

但是如果你有这样的需求，非要在控制台显示上传的文件内容，那么你可以使用fileItem.getString(“utf-8”)来处理编码。

**文本文件内容和普通表单项内容使用FileItem类的getString(“utf-8”)来处理编码。**

### 上传文件同名问题（文件重命名）

通常我们会把用户上传的文件保存到uploads目录下，但如果用户上传了同名文件呢？这会出现覆盖的现象。处理这一问题的手段是使用UUID生成唯一名称，然后再使用“\_”连接文件上传的原始名称。

例如用户上传的文件是“我的一寸照片.jpg”，在通过处理后，文件名称为：“891b3881395f4175b969256a3f7b6e10\_我的一寸照片.jpg”，这种手段不会使文件丢失扩展名，并且因为UUID的唯一性，上传的文件同名，但在服务器端是不会出现同名问题的。

|  |
| --- |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setCharacterEncoding("utf-8");  DiskFileItemFactory dfif = **new** DiskFileItemFactory();  ServletFileUpload fileUpload = **new** ServletFileUpload(dfif);  **try** {  List<FileItem> list = fileUpload.parseRequest(request);  //获取第二个表单项，因为第一个表单项是username，第二个才是file表单项  FileItem fileItem = list.get(1);  String name = fileItem.getName();//获取文件名称    // 如果客户端使用的是IE6，那么需要从完整路径中获取文件名称  **int** lastIndex = name.lastIndexOf("\\");  **if**(lastIndex != -1) {  name = name.substring(lastIndex + 1);  }    // 获取上传文件的保存目录  String savepath = **this**.getServletContext().getRealPath("/WEB-INF/uploads");  String uuid = CommonUtils.*uuid*();//生成uuid  String filename = uuid + "\_" + name;//新的文件名称为uuid + 下划线 + 原始名称    //创建file对象，下面会把上传文件保存到这个file指定的路径  //savepath，即上传文件的保存目录  //filename，文件名称  File file = **new** File(savepath, filename);    // 保存文件  fileItem.write(file);  } **catch** (Exception e) {  **throw** **new** ServletException(e);  }  } |

### 一个目录不能存放过多的文件（存放目录打散）

一个目录下不应该存放过多的文件，一般一个目录存放1000个文件就是上限了，如果在多，那么打开目录时就会很“卡”。你可以尝试打印C:\WINDOWS\system32目录，你会感觉到的。

也就是说，我们需要把上传的文件放到不同的目录中。但是也不能为每个上传的文件一个目录，这种方式会导致目录过多。所以我们应该采用某种算法来“打散”！

打散的方法有很多，例如使用日期来打散，每天生成一个目录。也可以使用文件名的首字母来生成目录，相同首字母的文件放到同一目录下。

日期打散算法：如果某一天上传的文件过多，那么也会出现一个目录文件过多的情况；

首字母打散算法：如果文件名是中文的，因为中文过多，所以会导致目录过多的现象。

我们这里使用hash算法来打散：

1. 获取文件名称的hashCode：int hCode = name.hashCode();；
2. 获取hCode的低4位，然后转换成16进制字符；
3. 获取hCode的5~8位，然后转换成16进制字符；
4. 使用这两个16进制的字符生成目录链。例如低4位字符为“5”

这种算法的好处是，在uploads目录下最多生成16个目录，而每个目录下最多再生成16个目录，即256个目录，所有上传的文件都放到这256个目录下。如果每个目录上限为1000个文件，那么一共可以保存256000个文件。

例如上传文件名称为：新建 文本文档.txt，那么把“新建 文本文档.txt”的哈希码获取到，再获取哈希码的低4位，和5~8位。假如低4位为：9，5~8位为1，那么文件的保存路径为uploads/9/1/。

|  |
| --- |
| **int** hCode = name.hashCode();//获取文件名的hashCode  //获取hCode的低4位，并转换成16进制字符串  String dir1 = Integer.*toHexString*(hCode & 0xF);  //获取hCode的低5~8位，并转换成16进制字符串  String dir2 = Integer.*toHexString*(hCode >>> 4 & 0xF);  //与文件保存目录连接成完整路径  savepath = savepath + "/" + dir1 + "/" + dir2;  //因为这个路径可能不存在，所以创建成File对象，再创建目录链，确保目录在保存文件之前已经存在  **new** File(savepath).mkdirs(); |

### 上传的单个文件的大小限制

限制上传文件的大小很简单，ServletFileUpload类的setFileSizeMax(long)就可以了。参数就是上传文件的上限字节数，例如servletFileUpload.setFileSizeMax(1024\*10)表示上限为10KB。

一旦上传的文件超出了上限，那么就会抛出FileUploadBase.FileSizeLimitExceededException异常。我们可以在Servlet中获取这个异常，然后向页面输出“上传的文件超出限制”。

|  |
| --- |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setCharacterEncoding("utf-8");  DiskFileItemFactory dfif = **new** DiskFileItemFactory();  ServletFileUpload fileUpload = **new** ServletFileUpload(dfif);  // 设置上传的单个文件的上限为10KB  fileUpload.setFileSizeMax(1024 \* 10);  **try** {  List<FileItem> list = fileUpload.parseRequest(request);  //获取第二个表单项，因为第一个表单项是username，第二个才是file表单项  FileItem fileItem = list.get(1);  String name = fileItem.getName();//获取文件名称    // 如果客户端使用的是IE6，那么需要从完整路径中获取文件名称  **int** lastIndex = name.lastIndexOf("\\");  **if**(lastIndex != -1) {  name = name.substring(lastIndex + 1);  }    // 获取上传文件的保存目录  String savepath = **this**.getServletContext().getRealPath("/WEB-INF/uploads");  String uuid = CommonUtils.*uuid*();//生成uuid  String filename = uuid + "\_" + name;//新的文件名称为uuid + 下划线 + 原始名称    **int** hCode = name.hashCode();//获取文件名的hashCode  //获取hCode的低4位，并转换成16进制字符串  String dir1 = Integer.*toHexString*(hCode & 0xF);  //获取hCode的低5~8位，并转换成16进制字符串  String dir2 = Integer.*toHexString*(hCode >>> 4 & 0xF);  //与文件保存目录连接成完整路径  savepath = savepath + "/" + dir1 + "/" + dir2;  //因为这个路径可能不存在，所以创建成File对象，再创建目录链，确保目录在保存文件之前已经存在  **new** File(savepath).mkdirs();    //创建file对象，下面会把上传文件保存到这个file指定的路径  //savepath，即上传文件的保存目录  //filename，文件名称  File file = **new** File(savepath, filename);    // 保存文件  fileItem.write(file);  } **catch** (Exception e) {  // 判断抛出的异常的类型是否为FileUploadBase.FileSizeLimitExceededException  // 如果是，说明上传文件时超出了限制。  **if**(e **instanceof** FileUploadBase.FileSizeLimitExceededException) {  // 在request中保存错误信息  request.setAttribute("msg", "上传失败！上传的文件超出了10KB！");  // 转发到index.jsp页面中！在index.jsp页面中需要使用${msg}来显示错误信息  request.getRequestDispatcher("/index.jsp").forward(request, response);  **return**;  }  **throw** **new** ServletException(e);  }  } |

### 上传文件的总大小限制

上传文件的表单中可能允许上传多个文件，例如：



有时我们需要限制一个请求的大小。也就是说这个请求的最大字节数（所有表单项之和）！实现这一功能也很简单，只需要调用ServletFileUpload类的setSizeMax(long)方法即可。

例如fileUpload.setSizeMax(1024 \* 10);，显示整个请求的上限为10KB。当请求大小超出10KB时，ServletFileUpload类的parseRequest()方法会抛出FileUploadBase.SizeLimitExceededException异常。

### 缓存大小与临时目录

大家想一想，如果我上传一个蓝光电影，先把电影保存到内存中，然后再通过内存copy到服务器硬盘上，那么你的内存能吃的消么？

所以fileupload组件不可能把文件都保存在内存中，fileupload会判断文件大小是否超出10KB，如果是那么就把文件保存到硬盘上，如果没有超出，那么就保存在内存中。

**10KB是fileupload默认的值，我们可以来设置它。**

**当文件保存到硬盘时，fileupload是把文件保存到系统临时目录，当然你也可以去设置临时目录。**



|  |
| --- |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setCharacterEncoding("utf-8");  DiskFileItemFactory dfif = **new** DiskFileItemFactory(1024\*20, **new** File("F:\\temp"));  ServletFileUpload fileUpload = **new** ServletFileUpload(dfif);    **try** {  List<FileItem> list = fileUpload.parseRequest(request);  FileItem fileItem = list.get(1);  String name = fileItem.getName();  String savepath = **this**.getServletContext().getRealPath("/WEB-INF/uploads");    // 保存文件  fileItem.write(path(savepath, name));  } **catch** (Exception e) {  **throw** **new** ServletException(e);  }  }    **private** File path(String savepath, String filename) {  // 从完整路径中获取文件名称  **int** lastIndex = filename.lastIndexOf("\\");  **if**(lastIndex != -1) {  filename = filename.substring(lastIndex + 1);  }    // 通过文件名称生成一级、二级目录  **int** hCode = filename.hashCode();  String dir1 = Integer.*toHexString*(hCode & 0xF);  String dir2 = Integer.*toHexString*(hCode >>> 4 & 0xF);  savepath = savepath + "/" + dir1 + "/" + dir2;  // 创建目录  **new** File(savepath).mkdirs();    // 给文件名称添加uuid前缀  String uuid = CommonUtils.*uuid*();  filename = uuid + "\_" + filename;    // 创建文件完成路径  **return** **new** File(savepath, filename);  } |

## 文件下载

### 通过Servlet下载1

被下载的资源必须放到WEB-INF目录下（只要用户不能通过浏览器直接访问就OK），然后通过Servlet完成下载。

在jsp页面中给出超链接，链接到DownloadServlet，并提供要下载的文件名称。然后DownloadServlet获取文件的真实路径，然后把文件写入到response.getOutputStream()流中。

download.jsp

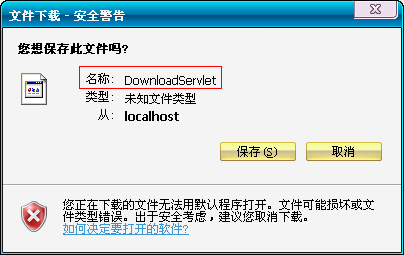
|  |
| --- |
| <body>  This is my JSP page. <br>  <a href=*"*<c:url value=*'/DownloadServlet?path=a.avi'*/>*"*>a.avi</a><br/>  <a href=*"*<c:url value=*'/DownloadServlet?path=a.jpg'*/>*"*>a.jpg</a><br/>  <a href=*"*<c:url value=*'/DownloadServlet?path=a.txt'*/>*"*>a.txt</a><br/>  </body> |

DownloadServlet.java

|  |
| --- |
| **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String filename = request.getParameter("path");  String filepath = **this**.getServletContext().getRealPath("/WEB-INF/uploads/" + filename);  File file = **new** File(filepath);  **if**(!file.exists()) {  response.getWriter().print("您要下载的文件不存在！");  **return**;  }  IOUtils.*copy*(**new** FileInputStream(file), response.getOutputStream());  } |

上面代码有如下问题：

* 可以下载a.avi，但在下载框中的文件名称是DownloadServlet；
* 不能下载a.jpg和a.txt，而是在页面中显示它们。



### 通过Servlet下载2

下面来处理上一例中的问题，让下载框中可以显示正确的文件名称，以及可以下载a.jpg和a.txt文件。

通过添加content-disposition头来处理上面问题。当设置了content-disposition头后，浏览器就会弹出下载框。

而且还可以通过content-disposition头来指定下载文件的名称！

|  |
| --- |
| String filename = request.getParameter("path");  String filepath = **this**.getServletContext().getRealPath("/WEB-INF/uploads/" + filename);  File file = **new** File(filepath);  **if**(!file.exists()) {  response.getWriter().print("您要下载的文件不存在！");  **return**;  }  response.addHeader("content-disposition", "attachment;filename=" + filename);  IOUtils.*copy*(**new** FileInputStream(file), response.getOutputStream()); |







　　虽然上面的代码已经可以处理txt和jpg等文件的下载问题，并且也处理了在下载框中显示文件名称的问题，但是如果下载的文件名称是中文的，那么还是不行的。

### 通过Servlet下载3

下面是处理在下载框中显示中文的问题！

其实这一问题很简单，只需要通过URL来编码中文即可！

download.jsp

|  |
| --- |
| <a href=*"*<c:url value=*'/DownloadServlet?path=这个杀手不太冷.avi'*/>*"*>这个杀手不太冷.avi</a><br/>  <a href=*"*<c:url value=*'/DownloadServlet?path=白冰.jpg'*/>*"*>白冰.jpg</a><br/>  <a href=*"*<c:url value=*'/DownloadServlet?path=说明文档.txt'*/>*"*>说明文档.txt</a><br/> |

DownloadServlet.java

|  |
| --- |
| String filename = request.getParameter("path");  // GET请求中，参数中包含中文需要自己动手来转换。  // 当然如果你使用了“全局编码过滤器”，那么这里就不用处理了  filename = **new** String(filename.getBytes("ISO-8859-1"), "UTF-8");    String filepath = **this**.getServletContext().getRealPath("/WEB-INF/uploads/" + filename);  File file = **new** File(filepath);  **if**(!file.exists()) {  response.getWriter().print("您要下载的文件不存在！");  **return**;  }  // 所有浏览器都会使用本地编码，即中文操作系统使用GBK  // 浏览器收到这个文件名后，会使用iso-8859-1来解码  filename = new String(filename.getBytes("GBK"), "ISO-8859-1");  response.addHeader("content-disposition", "attachment;filename=" + filename);  IOUtils.*copy*(**new** FileInputStream(file), response.getOutputStream()); |

## 文件上传模板

|  |
| --- |
| **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  // 1.服务器端保存上传文件的目录的绝对路径---upload目录在web服务器上的绝对路径  // 例如：D:\soft\tomcat\apache-tomcat-7.0.52\webapps\\uploadFileDemo\\upload  String savePath =**this**.getServletContext().getRealPath("/upload");  // System.out.println(savePath);  // 2.获取前台以multipart/form-data形式发送的 表单的所有参数  Collection<Part> parts = request.getParts();  // 遍历表单参数,每个Part对象就是一个表单元素，它可以是普通的文本输入框，也可以是文件域  **for** (Part part : parts) {  // 获取表单元素的name  String name = part.getName();  **if** (name.startsWith("file")) {  // 针对文件域进行分析,文件域中有个请求头content-disposition(获取Content-Disposition)  String header = part.getHeader("Content-Disposition");  // 值格式：form-data; name="file\_0"; filename="adv\_1.jpg"  // System.out.println(header);  String oldName = **this**.getOldName(header);  // 把文件名修改后再传到服务器上  String newName = **this**.getNewName(oldName);  // 把文件保存到服务器上,参数是文件保存在服务器上的绝对路径  part.write(savePath + "/" + newName);  }  }  }  // 根据content-disposition头的值来获取文件名  **public** String getOldName(String header) {  String[] arr = header.split(";");  // 截取出filename="xxx"这一部分  String filenameStr = arr[2];  filenameStr = filenameStr.substring(filenameStr.indexOf("\"") + 1,  filenameStr.length() - 1); // adv\_1.jpg  **return** filenameStr;  }  // 根据文件的原始名获取一个新文件名  **public** String getNewName(String oldName) {  // 获取后缀名(扩展名)  String ext = oldName.substring(oldName.lastIndexOf("."));  // 给文件改名两种方案：  // 1.使用UUID类产生唯一的字符串作为文件名,太长  // 2.使用当前的系统时间(精确到毫秒)作为文件名  Date d = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyyMMddHHmmssSSS");  String dateStr = sdf.format(d);  // 拼接新文件名  String newName = dateStr + ext;  **return** newName;  }  //使用uuid来创建一个随机的字符串。  **public** **static** **void** main(String[] args) {  **for** (**int** i = 0; i < 10; i++) {  // 用UUID类产生一个唯一的字符串 String str =  UUID.*randomUUID*().toString();  System.*out*.println(str);  }  } |

# Json（掌握）

## 什么是JSON

JSON(JavaScript Object Notation)：JS对象表示法，是一种轻量级的数据交换格式。

JSON是用字符串来表示Javascript对象，例如可以在Servlet中发送一个JSON格式的字符串给客户端Javascript，Javascript可以执行这个字符串，得到一个Javascript对象。

XML也可以用来为较大数据交换，前面已经学习过在Servlet中发送XML给Javascript，然后Javascript再去解析XML。

### JSON对象语法

JSON 语法：

对象：{ } 花括号阔起来的。

属性：使用双引号括起来。注：不能使用单引号。

JSON值：[数字](http://baike.baidu.com/view/37407.htm)（整数或浮点数）

字符串（在双引号中）

逻辑值（true 或 false）

数组[ ]（在方括号中）

对象（在花括号中）

null

规则：1并列的数据之间用逗号（“，”）分隔。

2映射用冒号（“：”）表示。

3 并列数据的集合（数组）用方括号("[]")表示。

4映射的集合（对象）用大括号（“{}”）表示。JSON对象

对象在js中表示为“{}”括起来的内容，数据结构为 {key：value,key：value,...}的键值对的结构，在面向对象的语言中，key为对象的属性，value为对应的属性值，所以很容易理解，取值方法为对象.key 获取属性值，这个属性值的类型可以是 数字、字符串、数组、对象几种。

比如：

①最简单的形式，可以用下面这样的 JSON 表示"名称 / 值对"：

{ "firstName": "Brett" } 相当于firstName=Brett

②多个"名称 / 值对"串

{ "firstName": "Brett", "lastName":"McLaughlin", "email": "aaaa" }

### 把Json字符串转换成Json对象(js的方法)

由后台传过来的为json字符串转位js对象。

1、JSON.parse()，它可以将JSON字符串转为js对象。它需要一个json字符串作为参数，会将该字符串转为JS对象。

例如：

|  |
| --- |
| var a = '{"address":"美国","age":"12"}';//json字符串  var b = JSON.parse(a);  alert(b.address);//美国  alert(b.age);//12 |

上述的JSON方法在ie7以下不适用可以使用eval函数

eval():可以用来执行字符串形式的代码，并将执行结果返回。

注：如果使用eval()执行的字符串中含有{ }，他会将{} 当成是代码块。如果不希望将其解析为代码块解析，则需要在字符串前后各加一个（）

|  |
| --- |
| var a = '{"address":"美国","age":"12"}';//json字符串  var object =eval("("+a+")");  console.log(object); |

eval()这个函数的功能很强大，可以直接执行一个字符串中的js代码，但是在开发中尽量不要使用，首先它的执行性能比较差，然后它还具有安全隐患，该方法可以形参的代码，可能有人会专门插入恶意的代码。

### JS对象转JSON

方法：JSON.stringify()

可以将一个JS对象转换为JSON字符串。需要一个js对象作为参数，会返回一个JSON字符串。

|  |
| --- |
| var a = {address:"美国",age:"12"};  var str = JSON.stringify(a);  console.log(str); // "{"address":"美国","age":"12"}" |

### 普通字符串，json字符串和json对象的区别。

Json和Js对象格式一样，只不过JSON字符串中的属性名必须加双引号，其他格式与JS语法一样。

有时候在做项目的时候时常将这两个概念弄混淆，尤其是在使用springmvc的时候，后台@RequestBody接受的是一个json格式的字符串，一定是一个字符串。先介绍一下json对象，首先说到对象的概念，对象的属性是可以用：对象.属性进行调用的。例如：

|  |
| --- |
| var person={"name":"zhangsan","sex":"男","age":"24"}//json对象  alert(person.name);//zhangsand  alert(typeof person);//object |

person就是json对象。可以用perosn.name这种方式进行属性的调用。第三行代码就是看person的类型，为object类型。

JSON字符串：我们常说的JavaScript中的字符串是单引号或者双引号引起来的。

|  |
| --- |
| var person='{"name":"zhangsan","sex":"男","age":"24"}';//json字符串  alert(person);//{"name":"zhangsan","sex":"男","age":"24"}  alert(typeof person);//string |

person就是一个json字符串，之所以叫json字符串，因为字符串的格式符合json的格式，第三行代码也匹配其中的类型为string。

### 数组

数组在js中是中括号“[]”扩起来的内容，数据结构为 ["java","javascript","vb",...]，取值方式和所有语言中一样，使用索引获取，字段值的类型可以是数字、字符串、数组、对象几种。

①含有1个属性的数组：

{ "people": [//属性

{ "firstName": "Brett", "lastName":"McLaughlin", "email": "aaaa" },属性值

{ "firstName": "Jason", "lastName":"Hunter", "email": "bbbb"},属性值

{ "firstName": "Elliotte", "lastName":"Harold", "email": "cccc" }属性值

]

}

在这个示例中，只有一个名为 people的变量，值是包含三个条目的数组，每个条目是一个人的记录，其中包含名、姓和email。

②含有多个变量的数组：

{ "programmers": [属性

{ "firstName": "Brett", "lastName":"McLaughlin", "email": "aaaa" }属性值

{ "firstName": "Jason", "lastName":"Hunter", "email": "bbbb" },属性值

{ "firstName": "Elliotte", "lastName":"Harold", "email": "cccc" }属性值

],

"authors": [属性

{ "firstName": "Isaac", "lastName": "Asimov", "genre": "science fiction" },属性值

{ "firstName": "Tad", "lastName": "Williams", "genre": "fantasy" },属性值

{ "firstName": "Frank", "lastName": "Peretti", "genre": "christian fiction" }属性值

]

}

### 例一

|  |
| --- |
| **var** person = {"name":"zhangSan", "age":"18", "sex":"male"};  alert(person.name + ", " + person.age + ", " + person.sex); |

　　注意，key也要在双引号中！

|  |
| --- |
| **var** person = {"name":"zhangSan", "age":"18", "sex":"male", "hobby":["cf", "sj", "ddm"]};  alert(person.name + ", " + person.age + ", " + person.sex + ", " + person.hobby); |

带有方法的JSON对象：

|  |
| --- |
| **var** person = {"name":"zhangSan", "getName":**function**() {**return this**.name;}};  alert(person.name);  alert(person.getName()); |

### JSON与XML比较

* 可读性：XML胜出；
* 解码难度：JSON本身就是JS对象（主场作战），所以简单很多；
* 流行度：XML已经流行好多年，但在AJAX领域，JSON更受欢迎。

### 把Java对象转换成JSON对象

apache提供的json-lib小工具，它可以方便的使用Java语言来创建JSON字符串。也可以把JavaBean转换成JSON字符串。

### jd对象和json对象的区别

## Json在Ajax中的应用

客户端可以给服务器端通过地址栏或者post很容易的提交数据，但是服务器端处理完数据之后，将计算的结果信息回传给客户端时就存在了一定的难度，特别是 数据量较大时。这个时候数据的格式成了关键，按照某种格式可以很方便的进行数据的组装，然后可以很方便的进行解析。使用Json便是一种很好的策略。在服务器端，按照Json的格式拼装好一个字符串，响应给客户端。客户端如何进行解析呢？一般有两种策略（两种策略的名称是自己给的名字，不一定很合理，但是 思路应该是没有问题的）：

### 直接解析

var json = eval('(' + result + ')');

eval(string): eval() 函数可计算某个字符串，并执行其中的的 JavaScript 代码。

### 间接解析

var json = "r=" + result;

eval(json);

当然上面行代码可以合并为：eval("r=" + result);

通过上面的计算，也可以将服务器端响应给客户端的Json格式的字符串解析成了一个Json（格式的）对象，但是该对象名称为“r”，通过“r.”或者“r[]”的方式可进行数据访问。

总结：Json是一种简单的数据交换格式，它几乎可以很好的代替xml让服务器之间灵活的交换数据。

## json-lib核心jar包

json-lib的核心jar包有：

* json-lib.jar

json-lib的依赖jar包有：

* commons-lang.jar
* commons-beanutils.jar
* commons-logging.jar
* commons-collections.jar
* ezmorph.jar

## json-lib中的核心类

在json-lib中只有两个核心类：

* JSONObject；
* JSONArray；4.1　JSONObject

JSONObject类本身是一个Map，所以学习它很方便。

|  |
| --- |
| JSONObject jo = **new** JSONObject();  jo.put("name", "zhangSan");  jo.put("age", "18");  jo.put("sex", "male");  System.*out*.println(jo.toString()); |

|  |
| --- |
| Person person = **new** Person("liSi", 18, "female");  JSONObject jo = JSONObject.*fromObject*(person);  System.*out*.println(jo.toString()); |

|  |
| --- |
| Map map = **new** HashMap();  map.put("name", "wangWu");  map.put("age", "81");  map.put("sex", "male");    JSONObject jo = JSONObject.*fromObject*(map);  System.*out*.println(jo.toString()); |

|  |
| --- |
| String xml = "<person><name>zhaoLiu</name><age>59</age><sex>female</sex></person>";  XMLSerializer serial = **new** XMLSerializer();  JSONObject jo = (JSONObject)serial.read(xml);  System.*out*.println(jo.toString()); |

## JSONArray

JSONArray本身是一个List，所以使用起来很方便。

|  |
| --- |
| JSONArray ja = **new** JSONArray();  Person p1 = **new** Person("zhangSan", 18, "male");  Person p2 = **new** Person("liSi", 23, "female");  ja.add(p1);  ja.add(p2);    System.*out*.println(ja.toString()); |

|  |
| --- |
| Person p1 = **new** Person("zhangSan", 18, "male");  Person p2 = **new** Person("liSi", 23, "female");  List<Person> list = **new** ArrayList<Person>();  list.add(p1);  list.add(p2);    JSONArray ja = JSONArray.*fromObject*(list);    System.*out*.println(ja.toString()); |

|  |
| --- |
| Person p1 = **new** Person("zhangSan", 18, "male");  Person p2 = **new** Person("liSi", 23, "female");  Person[] persons = {p1, p2};    JSONArray ja = JSONArray.*fromObject*(persons);    System.*out*.println(ja.toString()); |

## JS解释服务器发送过来的JSON字符串

服务器发送过来JSON字符串后，客户端需要对其进行解析。这时客户端需要使用eval()方法对JSON字符串进行执行！但要注意，eval()方法在执行JSON时，必须把JSON字符串使用一对圆括号括起来。

|  |
| --- |
| **var** json = "{\"name\":\"zhangSan\", \"age\":\"18\", \"sex\":\"male\"}";  **var** person = eval("(" + json + ")");  alert(person.name + ", " + person.age + ", " + person.sex); |

## json-lib解析json

### json文本转json对象

|  |
| --- |
| String jsonStr = "{\"address\":\"北京市\",\"id\":1001,\"name\":\"Tom\"}";  **try** {  // 1.把json文本包装成JSONObject  JSONObject jsonObj = **new** JSONObject(jsonStr);  // 2.调用JSONObject的getXxx(String key)来获取相应的value  String address = jsonObj.getString("address");  **int** id = jsonObj.getInt("id");  String name = jsonObj.getString("name");  // 3. 把上面的数据封装到javaBean对象上  Person p = **new** Person(id, name, address);  System.*out*.println(p);  } **catch** (JSONException e) {  e.printStackTrace();  } |

### 解析json对象

|  |
| --- |
| String jsonStr = "{\"person1\":{\"address\":\"深圳市\",\"id\":1001,\"name\":\"Tom\"}}";  **try** {  // 1.获取key为person1对应的value  JSONObject jsonObj = **new** JSONObject(jsonStr)  .getJSONObject("person1");  // 2.调用JSONObject的getXxx(String key)来获取相应的value  String address = jsonObj.getString("address");  **int** id = jsonObj.getInt("id");  String name = jsonObj.getString("name");  // 3. 把上面的数据封装到javaBean对象上  Person p = **new** Person(id, name, address);  System.*out*.println(p);  } **catch** (JSONException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } |

### 解析数组：把json文本转换为java数组或集合

|  |
| --- |
| String jsonStr = "[{\"id\":1001,\"name\":\"Tom\",\"address\":\"北京市\"},"  + "{\"id\":1002,\"name\":\"Mary\",\"address\":\"上海市\"}]";  **try** {  // 1.把json文本包装成JSON数组  JSONArray jsonArr = **new** JSONArray(jsonStr);  // list集合  List<Person> list = **new** ArrayList<Person>();  // Person数组  Person[] arr = **new** Person[jsonArr.length()];  // 2.遍历JSON数组  **for** (**int** i = 0; i < jsonArr.length(); i++) {  JSONObject jsonObj = (JSONObject) jsonArr.get(i);  String address = jsonObj.getString("address");  **int** id = jsonObj.getInt("id");  String name = jsonObj.getString("name");  Person p = **new** Person(id, name, address);  // 把javaBean对象添加到集合中  list.add(p);  // 加到数组中  arr[i] = p;  }  // 输出每个javaBean对象  **for** (Person person : list) {  System.*out*.println(person);  }  System.*out*.println("..........................");  **for** (Person person : arr) {  System.*out*.println(person);  } |

### 生成json--把对象或集合转换为json文本

|  |
| --- |
| public void test4() {  Person p1 = new Person(1, "孙悟空", "花果山");  Person p2 = new Person(2, "猪八戒", "高老庄");  List<Person> list = new ArrayList<Person>();  list.add(p1);  list.add(p2);  // 把对象转换我json文本  JSONObject jsonObj = new JSONObject(p1);  System.out.println(jsonObj);  // 把集合转换我json文本  JSONArray jsonArr = new JSONArray(list);  System.out.println(jsonArr.toString());  } |

## fastjson

### 解析对象--把json文本转换为javaBean对象

|  |
| --- |
| **public** **void** test1() {  String jsonStr = "{\"address\":\"北京市\",\"id\":1001,\"name\":\"Tom\"}";  Person p = JSON.*parseObject*(jsonStr, Person.**class**);  System.*out*.println(p);  } |

### 解析对象

|  |
| --- |
| **public** **void** test2() {  String jsonStr = "{\"person1\":{\"address\":\"深圳市\",\"id\":1001,\"name\":\"Tom\"}}";  // 先封装外层的JSONObject  JSONObject jsonObj = JSON.*parseObject*(jsonStr);  // 封装内层的JSONObject  JSONObject obj = jsonObj.getJSONObject("person1");  // 将内层的JSONObject变成字符串，再转换为对应的javaBean对象  Person p = JSON.*parseObject*(obj.toString(), Person.**class**);  System.*out*.println(p);  } |

### 解析数组--把json文本转换为javaBean对象的集合或数组

|  |
| --- |
| **public** **void** test3() {  String jsonStr = "[{\"id\":1001,\"name\":\"Tom\",\"address\":\"北京市\"},{\"id\":1002,\"name\":\"Mary\",\"address\":\"上海市\"}]";  List<Person> list = JSON.*parseArray*(jsonStr, Person.**class**);  **for** (Person person : list) {  System.*out*.println(person);  }  } |

### 生成json--把对象或集合转换为json文本

|  |
| --- |
| Person p1 = **new** Person(1, "孙悟空", "花果山");  Person p2 = **new** Person(2, "猪八戒", "高老庄");  List<Person> list = **new** ArrayList<Person>();  list.add(p1);  list.add(p2);  // 把对象转换为json文本  String objStr = JSON.*toJSONString*(p1);  // 把集合转换为json文本  String listStr = JSON.*toJSONString*(list); |

## Gjson

### 解析对象--把json文本转换为javaBean对象

|  |
| --- |
| **public** **void** test1() {  String jsonStr = "{\"address\":\"北京市\",\"id\":1001,\"name\":\"Tom\"}";  // 创建Gson对象  Gson gson = **new** Gson();  Person p = gson.fromJson(jsonStr, Person.**class**);  System.*out*.println(p);  } |

### 解析数组--把json文本转换为javaBean对象的集合或数组

|  |
| --- |
| **public** **void** test3() {  String jsonStr = "[{\"id\":1001,\"name\":\"Tom\",\"address\":\"北京市\"},{\"id\":1002,\"name\":\"Mary\",\"address\":\"上海市\"}]";  // 创建Gson对象  Gson gson = **new** Gson();  List<Person> list = gson.fromJson(jsonStr,  **new** TypeToken<List<Person>>() {  }.getType());  **for** (Person person : list) {  System.*out*.println(person);  }  } |

### 生成json--把对象或集合转换为json文本

|  |
| --- |
| **public** **void** test5() {  Person p1 = **new** Person(1, "孙悟空", "花果山");  Person p2 = **new** Person(2, "猪八戒", "高老庄");  List<Person> list = **new** ArrayList<Person>();  list.add(p1);  list.add(p2);  // 创建Gson对象  Gson gson = **new** Gson();  String objStr = gson.toJson(p1);  String listStr = gson.toJson(list);  System.*out*.println(objStr);  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println(listStr);  } |

# AJAX

## AJAX概述

### 什么是AJAX

AJAX（Asynchronous Javascript And XML）翻译成中文就是“异步Javascript和XML”。即使用Javascript语言与服务器进行异步交互，传输的数据为XML（当然，传输的数据不只是XML）。

AJAX还有一个最大的特点就是，当服务器响应时，不用刷新整个浏览器页面，而是可以局部刷新。这一特点给用户的感受是在不知不觉中完成请求和响应过程。

与服务器异步交互；

浏览器页面局部刷新；

响应数据的格式：text:纯文本

xml：

json：js提供的数据交换格式(主流)通过字符串来传递。

### 同步交互与异步交互

同步交互：

特点：1、客户端发出一个请求后，需要等待服务器响应结束后，才能发出二个请求；

2、刷新整个页面。

异步交互：

特点：1、客户端发出一个请求后，无需等待服务器响应结束，就可以发出第二个请求。

2、可以使用js来接收服务器的响应，然后使用js来局部刷新。

### AJAX常见应用情景



当我们在百度中输入一个“传”字后，会马上出现一个下拉列表！列表中显示的是包含“传”字的10个关键字。

其实这里就使用了AJAX技术！当文件框发生了输入变化时，浏览器会使用AJAX技术向服务器发送一个请求，查询包含“传”字的前10个关键字，然后服务器会把查询到的结果响应给浏览器，最后浏览器把这10个关键字显示在下拉列表中。

整个过程中页面没有刷新，只是刷新页面中的局部位置而已！

当请求发出后，浏览器还可以进行其他操作，无需等待服务器的响应！



当输入用户名后，把光标移动到其他表单项上时，浏览器会使用AJAX技术向服务器发出请求，服务器会查询名为zhangSan的用户是否存在，最终服务器返回true表示名为zhangSan的用户已经存在了，浏览器在得到结果后显示“用户名已被注册！”。  
 整页面没有刷新，只是局部刷新了；

在请求发出后，浏览器不用等待服务器响应结果就可以进行其他操作；

### AJAX的优缺点

优点：

AJAX使用Javascript技术向服务器发送异步请求；

AJAX无须刷新整个页面；

因为服务器响应内容不再是整个页面，而是页面中的局部，所以AJAX性能高；

ajax它的数据量小——因为他不重新加载整个页面（加载部分）

Ajax因为数据量小，响应速度快，用户体验好。

缺点：

AJAX并不适合所有场景，很多时候还是要使用同步交互；

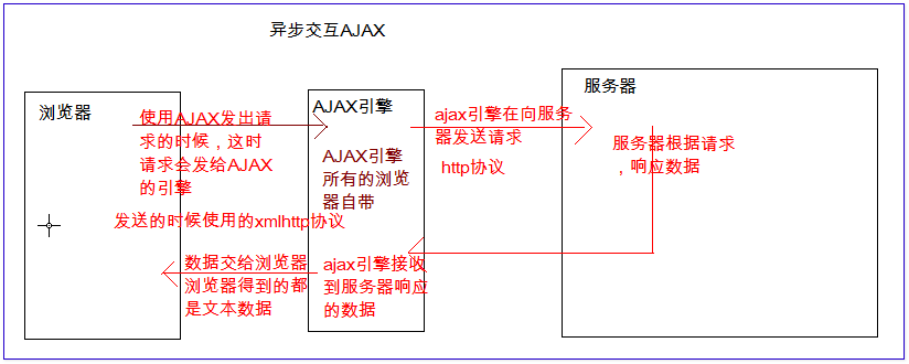
AJAX虽然提高了用户体验，但无形中向服务器发送的请求次数增多了，导致服务器压力增大；

因为AJAX是在浏览器中使用Javascript技术完成的，所以还需要处理浏览器兼容性问题；

### ajax的运行机制

在页面不刷新的情况下，向服务器发送请求，达到页面和后台的异步交互。

现在主流(IE、谷歌、火狐，其他的国产浏览器一般都是使用谷歌浏览器内核)的浏览器都有ajax引擎实现——现在ajax技术，都被主流浏览器实现，我们自己不用去写Ajax引擎，这个引擎已经存在在浏览器中。我们可以理解为浏览器都内置有ajax的核心对象，我们不需要自己创建核心对象，只要获取对象使用即可；



## AJAX技术

### 准备工作

因为AJAX也需要请求服务器，异步请求也是请求服务器，所以我们需要先写好服务器端代码，即编写一个Servlet！

这里，Servlet很简单，只需要输出“Hello AJAX!”。

|  |
| --- |
| **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  System.*out*.println("Hello AJAX!");  response.getWriter().print("Hello AJAX!");  }  } |

### AJAX核心（XMLHttpRequest）

其实AJAX就是在Javascript中多添加了一个对象：XMLHttpRequest对象。所有的异步交互都是使用XMLHttpRequest对象完成的。也就是说，我们只需要学习一个Javascript的新对象即可。

注意，各个浏览器对XMLHttpRequest的支持也是不同的！大多数浏览器都支持DOM2规范，都可以使用：var xmlHttp = new XMLHttpRequest()来创建对象；但IE有所不同，IE5.5以及更早版本需要：var xmlHttp = new ActiveXObject(“Microsoft.XMLHTTP”)来创建对象；而IE6中需要：var xmlHttp = new ActiveXObject(“Msmxl2.XMLHTTP”)来创建对象；而IE7以及更新版本也支持DOM2规范。

为了处理浏览器兼容问题，给出下面方法来创建XMLHttpRequest对象：

|  |
| --- |
| **function** createXMLHttpRequest() {  **var** xmlHttp;  // 适用于大多数浏览器，以及IE7和IE更高版本  **try**{//此为固定代码，因为浏览器不同则获取对象的方式处理也是不同的  xmlHttp = **new** XMLHttpRequest();  } **catch** (e) {  // 适用于IE6  **try** {  xmlHttp = **new** ActiveXObject("Msxml2.XMLHTTP");  } **catch** (e) {  // 适用于IE5.5，以及IE更早版本  **try**{  xmlHttp = **new** ActiveXObject("Microsoft.XMLHTTP");  } **catch** (e){}  }  }  **return** xmlHttp;  }  //演示效果  //alert(getXHR());  https://images2015.cnblogs.com/blog/1145937/201705/1145937-20170515220328916-1933782808.png |

### 打开与服务器的连接（open方法）

当得到XMLHttpRequest对象后，就可以调用该对象的open()方法打开与服务器的连接了。open()方法的参数如下：

open(method, url, async)：

 method：请求方式，通常为GET或POST；

 url：请求的服务器地址，例如：/ajaxdemo1/AServlet，若为GET请求，还可以在URL后追加参数；

 async：这个参数可以不给，默认值为true，表示异步请求；

**var** xmlHttp = createXMLHttpRequest();

xmlHttp.open("GET", "/ajaxdemo1/AServlet", **true**);

### 发送请求

xmlHttp.send(null);

参数：就是请求体的内容！如果是GET请求，必须给出null；

当使用open打开连接后，就可以调用XMLHttpRequest对象的send()方法发送请求了。send()方法的参数为POST请求参数，即对应HTTP协议的请求体内容，若是GET请求，需要在URL后连接参数。

注意：若没有参数，需要给出null为参数！若不给出null为参数，可能会导致FireFox浏览器不能正常发送请求！

|  |
| --- |
| xmlHttp.send(**null**); |

### 接收服务器响应

当请求发送出去后，服务器端Servlet就开始执行了，但服务器端的响应还没有接收到。接下来我们来接收服务器的响应。

XMLHttpRequest对象有一个onreadystatechange事件，它会在XMLHttpRequest对象的状态发生变化时被调用。下面介绍一下XMLHttpRequest对象的5种状态：

* 0：初始化未完成状态，只是创建了XMLHttpRequest对象，还未调用open()方法；
* 1：请求已开始，open()方法已调用，但还没调用send()方法；
* 2：请求发送完成状态，send()方法已调用；
* 3：开始读取服务器响应；
* 4：读取服务器响应结束。(通常我们只关心最后这个状态)

onreadystatechange事件会在状态为1、2、3、4时引发。

　　下面代码会被执行四次！对应XMLHttpRequest的四种状态！

|  |
| --- |
| xmlHttp.onreadystatechange = **function**() {  alert('hello');  }; |

但通常我们只关心最后一种状态，即读取服务器响应结束时，客户端才会做出改变。我们可以通过XMLHttpRequest对象的readyState属性来得到XMLHttpRequest对象的状态。

|  |
| --- |
| xmlHttp.onreadystatechange = **function**() {  **if**(xmlHttp.readyState == 4) {  alert('hello');  }  }; |

其实我们还要关心服务器响应的状态码是否为200，其服务器响应为404，或500，那么就表示请求失败了。我们可以通过XMLHttpRequest对象的status属性得到服务器的状态码。

最后，我们还需要获取到服务器响应的内容，可以通过XMLHttpRequest对象的responseText得到服务器响应内容。

|  |
| --- |
| xmlHttp.onreadystatechange = **function**() {  **if**(xmlHttp.readyState == 4 && xmlHttp.status == 200) {  //双重判断，只有4状态，同时响应成功。  alert(xmlHttp.responseText);//得到文本格式内容  }  }; |

## js实现ajax

|  |
| --- |
| function createXMLHttpRequest() {  var xmlHttp;  // 适用于大多数浏览器，以及IE7和IE更高版本  try{  xmlHttp = new XMLHttpRequest();  } catch (e) {  // 适用于IE6  try {  xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");  } catch (e) {  // 适用于IE5.5，以及IE更早版本  try{  xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");  } catch (e){}  }  }  return xmlHttp;  }  var btn = document.getElementById("btn");  btn.onclick = function (){  var xmlHttp = createXMLHttpRequest();  xmlHttp.open("GET", "/ajax/AServlet", true);  xmlHttp.send(null);  xmlHttp.onreadystatechange = function() {  if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {  var text = xmlHttp.responseText;  var h1 = document.getElementById("h1");  h1.innerHTML = text;  };  };  }; |

## Jquery实现ajax

### $.ajax()

|  |
| --- |
| //检测用户名  <input type=*"button"* value=*"点我试试"* onclick="loadData()">  <!-- 引入jQuery文件 -->  <script type=*"text/javascript"* src=*"js/jquery.min.js"*></script>  <script type=*"text/javascript"*>  **function** loadData(){  //发送异步请求  $.ajax({  //要请求的服务器资源的路径  url:'test.action',  //提交请求的方法：post或get， 不区分大小写  type:'POST',  //data向服务器发送的参数数据，可以写成json格式{key:value}，也可写成key= value&key2=valu2...  data: {"name":zhangsan,"age":20 },  //dataType:服务器端返回的数据格式，有json/html/xml/text  dataType:'text',  //success:是指定服务器端处理成功时调用的函数，参数data表示服务器端响应回来的数据(返回的数据)  success:**function**(data){  console.log("处理成功");  console.log(data);  },  //complete:指定服务器端处理完成时调用的函数，不管成功或失败都会调用的函数  //参数1：ajax底层的XMLHttpRequest对象,参数2：成功或失败的字符串  complete:**function**(xhr,text){  console.log("处理完成");  console.log(xhr);  console.log(text);  }  });  }  </script> |

### $.post

格式：$.post(url,[data],[callback],[type]))

url：string类型，ajax请求的地址

data：可选参数，object类型，发送至服务器的key/value数据会作为QueryString附加到请求URL中。

callback：可选参数，function类型，当ajax返回成功时自动调用该函数。

|  |
| --- |
| **function** checkUname(){  //1.获取输入的用户名  **var** uname= $('#uname').val();  //2.发送异步请求  $.post('user.action?method=checkUname',{"uname":uname },**function**(result){  **if**(result=="1"){  //已存在  $('#msg').html('<font color="red">用户名已存在，请更换！</font>');  }**else**{  //不存在  $('#msg').html('<font color="green">√</font>');  }  });  } |

注：

1)如果客户端发异步请求时，没有请求参数，则data属性可以省略

2)dataType:”text”表示客户端接收到的是服务器发回来的文本，此时会将服务器端响应回来的数据转换为string类型

3)写请求参数时，格式可以是json格式{key:value}，也可写成"key= value&key2=valu2..."

例如：

Json格式：data: {"name":"zhangsan","age":20 },

&格式：data : "name=zhangsan&age=22"

4).当dataType为json时，$.ajax方法能够将后台返回的json文本数据转换成object ，即json对象形式{key:value,key2:value}，读取对象的数据直接用success函数中的参数data.key。

|  |
| --- |
| success : **function**(data) {  console.log("success");  //当dataType是json时，$.ajax有能力自动将返回的json文本转换成object  console.log(**typeof** data);  // 把返回的数据通过js+dom+css显示在网页上  $('#msg').html("姓名："+data.uname+",年龄："+data.age);  // console.log(data);  }, |

### 如何选$.post()或$.ajax()?

1.$.post()方法适合于简单的异步交互，即服务器端返回text数据，

2. $.ajax()方法更强大，能处理服务器端返回json的数据

不管后台返回什么数据，$.ajax()都可以处理

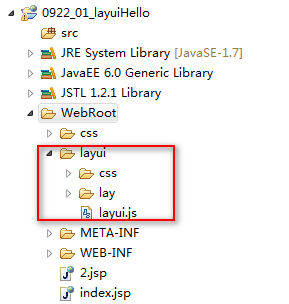
2.3 Ajax异步请求-$.get()

$.get()和$.post()方法的参数完全一样，只不过它发的是GET请求

## ajax提交表单

## layui

### 导包



### 页面引入

|  |
| --- |
| <!-- 引入layui的资源文件 -->  CSS文件  <link type=*"text/css"* rel=*"stylesheet "*href=*"layui/css/layui.css"*>  JS文件  <script type=*"text/javascript"* src=*"layui/layui.js"*></script> |

### 弹出层

|  |
| --- |
| <script type=*"text/javascript"*>  // 给li注册点击事件  **var** lis = $('.buttonValue li');  lis.click(**function**() {  //获取当前点击的li的rel属性值  **var** rel = $(**this**).attr("rel");  **switch** (rel) {  **case** '1':  //使用layui的layer模块  layui.use('layer', **function**() {  //调用open方法打开对应的弹出层  layer.open({  type : 0,  content : '我是信息层---传入任意的文本或html'  });  });  **break**;  **case** '2':  layui.use('layer', **function**() {  //调用open方法打开对应的弹出层  layer.open({  type : 1, //页面层  // content : '我是页面层---<h2>我是加粗的内容</h2>',//页面层的content是html字符串  content:'index.jsp',  area : [ '500px', '300px' ] //宽高    });  });  **break**;  **case** '3':  layui.use('layer', **function**() {  //调用open方法打开对应的弹出层  layer.open({  title:['优酷首页','color:red'], //标题，代替默认的“信息”  type : 2, //iframe层  // content: 'http://www.baidu.com' , //iframe成的content是一个url(资源的路径)  content : 'index.jsp',  area : [ '500px', '300px' ], //宽高  time: 15000, //自动关闭的时间，单位是毫秒  maxmin:**true**, //显示最大最小化按钮  anim:6, //动画，抖动  closeBtn:2, //关闭按钮，默认是1  });  });  **break**;  **case** '4':  layui.use('layer', **function**() {  //调用open方法打开对应的弹出层  layer.open({  title:['优酷首页','color:red'], //标题，代替默认的“信息”  type : 3, //加载层  // content: 'http://www.baidu.com' , //iframe成的content是一个url(资源的路径)  content : 'index.jsp' , //content的值也是字符串  });  });  **break**;  }  });  </script> |

# JSONP

## 同源策略

要理解跨域，先要了解一下“同源策略”。所谓同源是指，域名，协议，端口相同。所谓“同源策略“，简单的说就是基于安全考虑，当前域不能访问其他域的东西。

一些常见的是否同源示例可参照下表：



## JSONP

JSONP 是 JSON with padding（填充式 JSON 或参数式 JSON）的简写。

JSONP实现跨域请求的原理简单的说，就是动态创建<script>标签，然后利用<script>的src 不受同源策略约束来跨域获取数据。

JSONP 由两部分组成：回调函数和数据。

回调函数是当响应到来时应该在页面中调用的函数。

回调函数的名字一般是在请求中指定的。而数据就是传入回调函数中的 JSON 数据。

|  |
| --- |
| 1. $.ajax({ 2. async : true, 3. url : "https://api.douban.com/v2/book/search", 4. type : "GET", 5. dataType : "jsonp", *// 返回的数据类型，设置为JSONP方式* 6. jsonp : 'callback', *//指定一个查询参数名称来覆盖默认的 jsonp 回调参数名 callback* 7. jsonpCallback: 'handleResponse', *//设置回调函数名* 8. data : { 9. q : "javascript", 10. count : 1 11. }, 12. success: function(response, status, xhr){ 13. console.log('状态为：' + status + ',状态是：' + xhr.statusText); 14. console.log(response); 15. } 16. }); |

# 知识点

## 配置自动跳转首页

welcome-file：配置网站的欢迎页面，在访问欢迎页面时，无需在项目后写欢迎页面； 如果配置了多个欢迎页面，则jsp引擎从上往下在根目录下查找欢迎页是否存在，找到了就把它作为欢迎页

就不会再找了。 -->

在web.xml中添加如下：

<welcome-file-list>

<welcome-file>test.jsp</welcome-file>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

## jsp中自动跳转到某个页面

|  |
| --- |
| <script type="text/javascript">  //改变请求网址  window.location.href="test.jsp";  </script> |

## 表单的重复提交

## 修改数据库值时，里面的时间会自动变更

## 富文本编辑器

## 错误页面配置

在java web工程中，也可以对web.xml配置文件配置全局的错误页面配置，这样在工程中发生错误，都会跳转到相应的页面。一般是在项目上线的时候使用。

|  |
| --- |
| <!-- 全局错误处理页面配置 -->  <error-page>  <!--服务器发生错误时-->  <error-code>500</error-code>  <location>/errorPage.jsp</location>  </error-page>  <error-page>  <!--资源找不到-->  <error-code>404</error-code>  <location>/404.html</location>  </error-page> |

## 上传头像